Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Master in Statistics and Operations Research
Master's Thesis

Abstraction capabilities of deep neural networks while learning chess

Carles Mitjans Coma

Advisor: Alexandre Perera

B2SLab

Abstract

Key words: Machine learning, Deep Neural Networks, Convolutional Neural Networks, Chess

The increasing usage of machine learning techniques in research is a clear indicator of its usefulness in this area. Machine learning has been around for more than a couple decades now, and it continues to expand in new disciplines. Machine learning techniques are also growing, and new approaches are continuously being developed and studied. As a consequence, there is a lack of knowledge in the intricate workings of these new techniques. Nowadays, fully detailed programming libraries focused on machine learning algorithms (such as Keras, Tensorflow, and Torch among others) present the end user with a vast number of these new techniques. Thus, the understanding of how they work and how to use them is crucial for anyone willing to apply them in its own research area.

Among these new techniques, there are a few that are based around the same concept; that is, Neural Networks. Neural Networks have been around for some time now, and have also evolved to more complex approaches. As the name "Neural" implies, they are brain-inspired systems which are intended to replicate the way that humans learn. In order to better understand Neural Networks, they have been used in various tasks from different disciplines. The more intuition required by the task in which Neural Networks are used, the more complex and human-like is the learning curve for that task. Hence, trying to replicate the human intuition and learning algorithm in these tasks has been a challenge for a lot of researchers.

A well-known research team focused on solving human intelligence is DeepMind, the designers of AlphaGo. AlphaGo is a computer program that achieved superhuman intelligence in the game of Go, one of the most challenging board games of all time. While Go has been the latest milestone conquered by artificial intelligence and machine learning, Chess was once a notorious target for machine learning, as it also requires years of practice to master it.

In this thesis, different Neural Network and Deep Neural Network structures will be studied to see how they influence the learning curve for chess. It is worth noting that the main goal of this study is not to dominate the game of chess but to observe and study how Neural Networks learn to play it from the ground up and without prior knowledge of its rules.

Acknowledgements

First and foremost, I would like to thank Dr. Alenxandre Perera i Lluna for his support throughout this project at *Centre de Recerca en Enginyeria Biomèdica* (CREB). His mentoring and dedication to his students exceeded all expectations by far.

I would also like to thank all people currently researching at CREB for their kindness, creating a very comfortable place to work at. They were always willing to help me.

Next, I would like to thank Jordi Ventura for his help during the last phase of this thesis. He is the author of almost all figures presented in this paper. His help was crucial to make parts of this project more understandable and easy to read. *Moltes gràcies Jordi i espero poder-te ajudar de la mateixa manera amb els treballs i projectes que encara t'han de venir.*

A very special mention goes to my friends and colleagues at AEiG Sant Ignasi, specially Jordina Torrents and Roman Carreras. They have always been there when I have had a rough time, cheering for me and supporting me whenever possible. *Moltíssimes gràcies i per molts anys més! Visca el CAU!*

Finally, but not less important, a huge thank you goes to my family. Without them this project wouldn't have happened. *Moltes gràcies pels multiples consells que m'heu pogut donar durant el transcurs d'aquest treball.*

Contents

Chapter 1
Introduction

1. Mission statement and objectives

Following a large number of projects developed around the concept of Machine Learning and Neural Networks, this thesis seeks to shorten the gap between the intrinsics of Neural Networks and the end-user. In particular, the thesis will focus on two types of Neural Networks: Artificial Neural Networks and Convolutional Neural Networks.

Previous studies in this field try to optimize the algorithm to outperform human players in the game of Chess. To achieve this, the methods presented so far are not only based on Neural Networks but also on lookahead algorithms. These lookahead approaches in conjunction with Neural Networks that are capable of evaluating the board position, allow the resulting computer programs to surpass any human mind in the game of chess. Other studies try to minimize the usage of these lookahead algorithms, implementing more than one Neural Network to predict the next movement in the game. In this thesis, only one neural network will be used each time to predict the next movement. Although this approach may be inefficient, it help us better understand the underlying workings of Neural Networks.

The main objectives of the thesis can be summarized in the following points:

- Design, develop and program a framework to ease the communication between chess binary representations and neural networks.
- Devise different Neural Network structures that could potentially learn to play chess, although being able to beat a human player wasn't a requirement. Instead, studying how and why the neural networks learned to play chess was the main objective.
- Design and apply statistics to better discern which Neural Network structure works better and why.

2. Project scope

Board games have been a primary target for machine learning algorithms since its conception. Games like Three in a Row, Backgammon, Checkers, Othello, Chess and more recently Go have all been outplayed by programs. One of the reasons why board games have been the main goal for artificial intelligence and machine learning in the last decades is because of the intuition needed to play them. Human intuition is a clear manifestation of a background learning process, and it comes after many trials and errors while performing the same task.

While it is true that this intuition may be needed while executing other tasks, board games present an avail. Board games are defined by a very rigorous set of rules and have a very well-defined goal. This is a key aspect to take into consideration, as computers have a difficulty to understand things if they are not presented in a very specific manner. Also, all the information that may influence the outcome result can be harvested from the game position. Take as an example the task of forecasting the weather. Despite the huge amount of data we have on previous weather, the forecast nowadays isn't a 100% accurate and we can only estimate the weather for the next few days. This is because there is a tremendous amount of information we can't collect that affect the outcome. This issue doesn't appear in traditional board games. All the data that may affect the outcome of the game are present in the game itself.

Nonetheless, artificial intelligence and machine learning algorithms have evolved so much in the past years that nowadays the resulting computer programs tend to be very complex with multiple algorithms in use at once. A clear example of this is AlphaGo [**1**], the computer program developed by DeepMind back in 2016 that overcame the best Go player in the past years by 4 games against 1. AlphaGo was built upon 3 different kinds of neural networks in combination with a tree search procedure. Not only its computational structure was sophisticated but it also relied upon heavy hardware components. The first version of AlphaGo used 147 GPUs.

In contrast, this thesis will try to minimize the complexity and maximize the focus on the essentials of machine learning and neural networks. This project has been developed at B2SLab [**2**], a bioinformatics research group at UPC. The available hardware has been a couple of up-to-date GPUs to perform all the learning process. This has been a downside for this project, as the amount of time required to train the neural networks later presented has been a huge bottleneck during the project. Thus, the extent of neural networks studied had to be reduced to 5 after a couple of first tests. All of them have been programmed using the Python programming language in conjunction with Keras [**3**] and Tensorflow [**4**] machine learning libraries. Finally, R was used to perform statistics on the results gathered from the trained models.

3. Project Structure

The report is divided into two main parts: a theoretical part and a practical part. It starts with the theoretical one, explaining in detail the fundamentals and inner workings of Artificial Neural Networks and Convolutional Neural Networks. Especial attention will be given to the process of creation of such neural networks as well as the structure and learning process of them.

Once the theoretical part is finished, the practical part will start, in which the building process for the neural networks used in this project will be discussed. This part will start by introducing the binary representation used as the neural network's input. In this step, the information is represented in a way that the neural network can understand and learn from. Thus, this step is sometimes considered the most important in the flowchart of creating a neural network.

Next, focus will be given to the creation of a simple Artificial Neural Network and its structure, followed by the creation of a Convolutional Neural Network, for which the representation of a chess game had to be modified. It will also discuss the statistical approach to determine which neural networks performed better and why. This part will also briefly introduce Keras and Tensorflow, the two Python libraries used during this thesis for the development of such neural networks.

Finally, the last sections will be dedicated to the results and conclusions gathered from this project. A special section for future work will be created in which new and better machine learning approaches will be discussed, and how and why these new techniques could improve the methods presented here.

Part 1

Theoretical Framework

Chapter 2
The perceptron

1. Single perceptron

The perceptron, devised by Frank Rosenblatt in 1958 [**5**], represents a pivoting point for machine learning and neural networks. Nowadays, perceptrons are no longer used in complex networks but are the foundation from which modern techniques and types of neural structures are based. In order to understand more complex models and neurons, it's worth taking the time to first understand perceptrons.

To better picture a perceptron, it's ideal to imagine it as a single node in a more complex network of neural connections from the brain. As the name Neural Networks implies, the concept is based on how the brain is organized. A neuron in a biological system is a cell connected to multiple other neurons; some of these connections serve as inputs to the neuron and the others serve as outputs. A perceptron is designed to follow this pattern in a more simplistic manner: the perceptron has several binary inputs $(x_1, x_2, ..., x_n)$ and produces a single binary output (y).

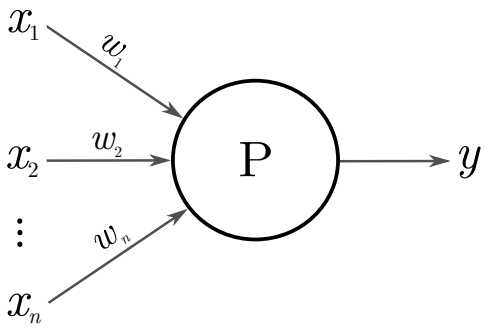


FIG. 1. A single perceptron is shown. A perceptron can have multiple inputs x_i, but always one output y. w_i is the weight for the ith input.

The example shown in Fig. 1 illustrates a perceptron with n inputs represented as x_i. To compute the output of the perceptron, Rosenblatt proposed a simple rule

based on weights. Each input has a corresponding weight w_i representing (again, based on a biological neural system) the relevance of that input for the perceptron. The output, then, is determined by whether the weighted sum $\sum x_i w_i$ of the inputs is higher or lower than a predefined threshold value. As the resulting value of the weighted sum is a real number, the more precise algebraic notation would be:

$$y = \begin{cases} 0, & \text{if } \sum x_i w_i \leq \text{threshold} \\ 1, & \text{if } \sum x_i w_i > \text{threshold} \end{cases} \tag{1}$$

It is important to notice that the weights w_i might have negative values, which would suggest that the corresponding input drives the perceptron to output a 0.

This is the basic mathematical model behind a perceptron. In a way, perceptrons can be thought of as machines that make decisions based on weighted inputted evidence. A general example would be a perceptron that tries to discern whether a person is a female or a male based on 3 factors. The first step would be to decide the binary representation for both males and females. In this example, males represent 0 and females 1. The decision will be made weighing up three factors:

- Does the person weight more than 60kg?
- Is the person's height greater than 160cm?
- Is the person's hair long(1) or short(0)?

Now, if we are working with a sample from which all males have short hair and all females have long hair, we could match the weight for the 3rd factor to be equal or greater than the threshold. In this case, any person with long hair will be classified as a female and it wouldn't make a difference whether a person weights more than 60Kg, for example.

By varying the weights and the threshold, different decision-making models can be obtained. This isn't a complete model of human decision-making, but it illustrates how a perceptron can be used to weight up evidence in order to make a decision.

2. Multi-layered perceptrons (MLP)

What if we then take the output from a perceptron and use it as an input to another? This is the next step in making a more complex model, and is represented in Fig. 2.

First of all, notice how the inputs have been encircled. This is a notation widely used in neural networks to represent what the *input layer* is. They do not represent new perceptrons or neurons in the net, but it's merely a notation.

The next layer, or first *hidden layer* (HL^0), is the first layer of the network that weights up the evidence from the input layer. This layer is a combination of various single perceptrons like the one previously seen. The output from this perceptrons is then used by a second hidden layer of perceptrons (HL^0). Notice how the evidence from which the hidden layer makes a decision is in turn the decision of a previous

layer. This can be thought as a chain of decisions based on previous decisions. Thus, the level of complexity at each layer increments, and more sophisticated models can be made.

In theory, the number of hidden layers and the number of perceptrons in each layer can be whatever number you would like. This, in turn, adds another level of complexity to the whole network, but the complexity added to the model doesn't correlate in a linear way to its accuracy.

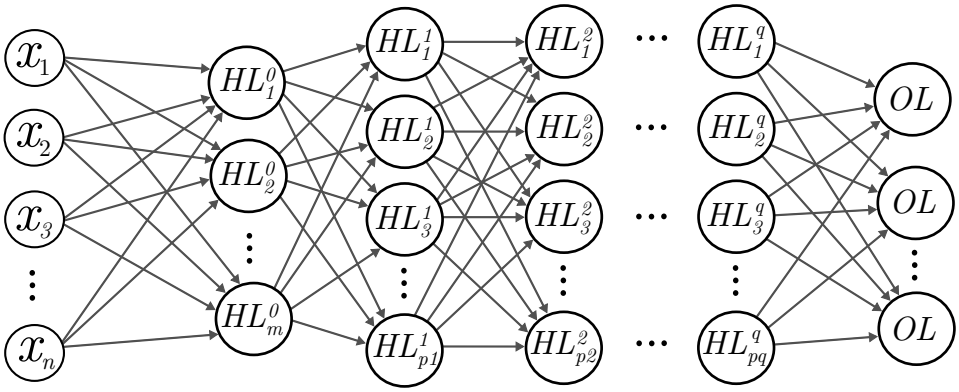


FIG. 2. Multi-Layered Perceptron (MLP). The number of layers and the number of neurons per layer can be anything. The more complex the model, the more computational impact it will have. x_i are the inputs. HL_i^j stands for the ith neuron in the jth hidden layer. OL stands for output layer. p_j is the number of neurons in the jth layer and q is the number of hidden layers.

In Fig. 2, a multi-layered perceptron is presented. It's important to notice that the output layer can have multiple perceptrons. This is useful in classification tasks: each neuron in the output layer specializes in a single class. This will be explained in more detail later in this project.

3. The bias

The algebraic equation presented before at Eq. 1 is a little bit cumbersome, and can be simplified as follows:

$$y = \begin{cases} 0, & \text{if } (x \cdot w) - \text{threshold} \leq 0 \\ 1, & \text{if } (x \cdot w) - \text{threshold} > 0 \end{cases} \tag{2}$$

where $x \cdot w \equiv \sum x_i w_i$, and x and w are vectors whose components are the inputs and weights, respectively. Moving the *threshold* to the other side of the inequality lets us understand it from a different perspective. Now the threshold can be thought

of as a measure that determines how easy it is for the perceptron to output a 1. If the threshold is a very low positive value and $w_i > 0 \ \forall i$ and at leat one input is non zero, the inequality will have higher chances to be true and the perceptron will activate. Technically, this concept is understood as the *bias* of a perceptron, and it is normally represented as a negative value, $b \equiv -\text{threshold}$. Hence, the algebraic equation can be rewritten as follows:

$$y = \begin{cases} 0, & \text{if } (x \cdot w) + b \leq 0 \\ 1, & \text{if } (x \cdot w) + b > 0 \end{cases} \tag{3}$$

A perceptron with a very high bias (b) will be very easy to activate and won't depend much on the inputs, whereas a perceptron with a very low bias will vary its output much more.

Chapter 3
Artificial Neural Networks

1. Learning process. The sigmoid neuron

Up until now, the perceptrons introduced cannot learn by themselves. You have to manually adjust the weights and biases in order for the network to perform well in a given task, hence, they are fixed models with no margin to change. In fact, another use of perceptrons is to compute the elementary logical functions widely used in computation (AND, OR, NAND, etc) [**6**]. For example, using a single perceptron with two binary inputs and the same weight for each one, implements a logical NAND gate, from which any other logical computation can be derived. These are fixed networks without any learning algorithm implemented.

The only two things that can be changed in a network are the biases (b) and the weights (w) of all perceptrons. Small changes in these values will produce a small change in the output (See Fig. 1). Hence, the goal of the learning algorithm is to correctly change the weights and the biases of the network to a value that will produce the best result overall. For example, suppose we are trying to recognize handwritten digits using a network of perceptrons. If the predicted value for an image with an "8" is "9", we can try to apply small changes to the biases and weights to certain perceptrons for the outcome to be more accurate.

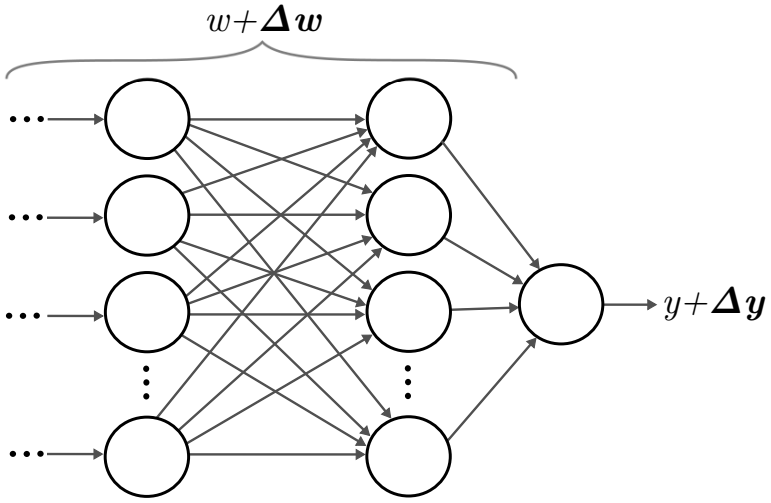


FIG. 1. This figure illustrates that small changes in the network's parameters produce a small change in the output of such. The network parameters are represented as w, although the biases are also parameters that can be tuned.

If it were true that small changes in a weight or bias cause a small change in the output, then we could use this fact to modify all weights and biases in a way that our network would perform best. Doing so repeatedly with new handwritten digits would produce better and better results over time.

There is a downside of doing this with the perceptrons introduced so far. As we are working with binary inputs and outputs and fixed thresholds for every perceptron, a small change in weight or bias in a single perceptron can cause the output of such perceptron to flip from 0 to 1. This small change can then cause the following layers of the network to produce a very different output from the one they were producing before. Thus, a small change in a weight or bias using this kind of perceptrons will produce a huge change in the output.

We can overcome this problem if we use a new kind of artificial neural neuron called the *sigmoid neuron*. There are a couple of changes from the original perceptron introduced by Rosenblatt. The first one is that the output of this neuron is a real value between 0 and 1. The second and more important change in this kind of neurons is the use of the sigmoid function as an activation function.

Instead of using a threshold that tells the neuron whether it should activate or not, the neuron uses an activation function (the sigmoid function). It uses the inputted weighted sum ($\sum x_i w_i$) as the argument to the sigmoid function which, in turn, produces the output y from the neuron. The sigmoid function is defined as follows:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{4}$$

Thus, the output of a neuron taking into account all inputted weights and biases will be:

$$y = \frac{1}{1 + e^{-\sum x_i w_i - b}} = \frac{e^{\sum x_i w_i + b}}{e^{\sum x_i w_i + b} + 1} \tag{5}$$

The sigmoid neuron is in fact quite similar to the original perceptron. Let's suppose that the $x \cdot w + b$ is a very large positive value. Then, $exp(x \cdot w + b) \approx +\infty$ and $S(x) \approx 1$, which would be the same output given by the perceptron. If instead, $x \cdot w + b$ is a very large negative value, $exp(x \cdot w + b) \approx 0$ and $S(x) \approx 0$ which, again, would be the same output from the perceptron. It seems, then, that the sigmoid function and the step function (the one used by the perceptron and defined by the threshold) behave the same way in their limits. They, instead, differ much when the values are relatively small in absolute value. This can be seen in Fig. 2, where the step function (centered at 0) and the sigmoid function are compared.

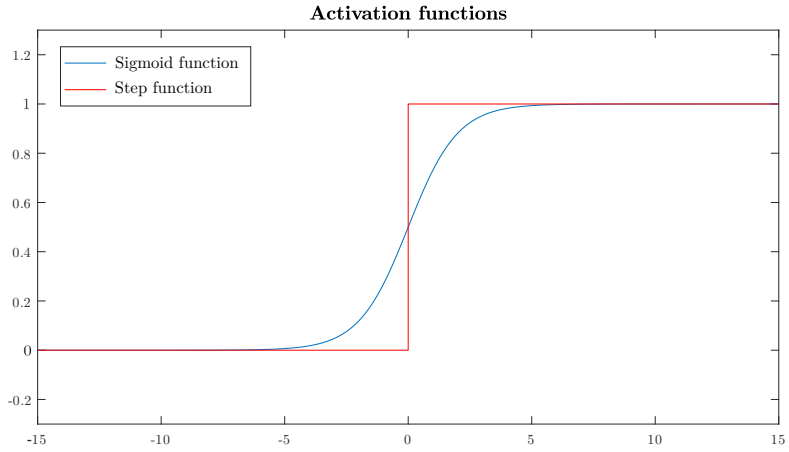


FIG. 2. The sigmoid function and the step function are compared. Both behave in the same way towards the limits.

Knowing this we can now define the activation function not only of the sigmoid neuron but also of the perceptron.

- Perceptron: step function
- Artificial neural neuron: sigmoidal function

2. Other types of activation functions

Activation functions are generally non-linear transformations of the weighted sum of the inputs to the neuron plus a bias. Thus, there are infinite possibilities for activation functions, although there are only a few that are widely used nowadays.

Now that we understand the basic concept behind the learning process of a neuron in a neural network, it is time to discuss other types of activation functions that have been developed more recently.

First of all, let's recall how the input to the activation function is calculated:

$$Y = \sum x_i w_i + bias \tag{6}$$

Where x_i is the *ith* input to the neuron and w_i is its weight. The value Y can be anything from $-\infty$ to $+\infty$ which makes it difficult to decide at which point the neuron should activate. From the previous section, the sigmoid function has been introduced, which was the first activation function to become widely used. The sigmoid function is a non-linear transformation that shifts the infinite bounds of the value Y to the bounds $(0, 1)$, from which the final output of the neuron is generated. Hence, we can think of activation functions as a way of telling the next neurons whether the current neuron has been activated or not and thus, if they should take it into consideration when computing their respective outputs.

The two activation functions introduced so far are the step function and the sigmoid function. The step function transforms the value Y to 0 or 1 and becomes the output of the neuron. The downside of this, as introduced before, is that a small shift in a single neuron may produces a large change in output. The sigmoid function overcomes this problem by means of its flexibility: it can produce real values between 0 and 1. But why sigmoid function and not any other? To answer this, let's assume for a moment the inexistence of the sigmoid function. Let us review the pros and cons of different functions that can help to shape the neuron activity.

2.1. Linear function $[f(x) = ax]$**.** Using a linear function, we solve the hide-bound problem of the step function. A small change in a weight or bias always produces a small change in the neuron's output. The linear function, though, presents a couple of downsides that makes it impractical. If all neurons in the network activate using a linear function, this creates a chain of linear functions that eventually will activate (or not) the final neuron. The downside is that this chain of linear activations is itself a single linear activation. No matter how many hidden layers there are, as long as they are linear in nature, the final activation function in the output layer is nothing but a simple linear function of the input layer. This means that despite of the depth of the network, a single layer could produce the same output, and therefore the level of complexity of the network is the simplest. The second downside of this type of activation function relates to *Gradient Descent*, the algorithm most used to train neural networks that will be presented later in this thesis. In a few words, Gradient Descent depends on the derivative of the activation function to perform changes in weights and biases (and thus, making small changes towards learning). The problem is that the derivative of a linear function is a constant and does not depend on x. This means that Gradient Descent will apply constant changes to weights and biases, not depending on the input presented [**7**].

This is a huge downside in the learning process, that makes (in conjunction with the first downside presented) linear functions impractical in neural networks [1].

2.2. Sigmoid function $[S(x) = \frac{1}{1+e^{-x}}]$**.** The sigmoid function has already been presented in the previous section of this chapter. Now that the problems of the step function and why linear functions don't solve them are better understood, sigmoid function becomes a more intuitive solution. Firstly, because of its nonlinearity. This allows for multiple layers to be stacked. In addition, as commented previously, small changes in the input will produce small changes in the output, which makes the learning process possible. Another advantage of sigmoid functions in comparison to linear functions is the fact that the range of the sigmoid function is $(0, 1)$, whereas the range of the linear function is $(-\infty, +\infty)$. The sigmoid function, although it seems perfectly valid for learning, also presents a downside. When the weighted sum is small, the sigmoidal curve is steeper, which means that the learning process will bring the activations to either side of the curve. This is good because different weighted sum values will respond differently when in this range. The problem arises in both ends of the curve, where small changes in the weighted sum output produce almost the same output. This means that gradient is small or vanished (cannot make significant changes to the neuron because of the small difference), which makes the learning process very slow. When this happens, the network refuses to learn or does so very slowly.

2.3. Tanh function or hiperbolic tangent $[tanh(x) = \frac{2}{1+e^{-2x}} - 1]$**.** Tanh is another activation function that is currently being used. Tanh is very similar in shape to the sigmoid function (see Fig. 3). In fact, it is a scaled version of the sigmoid function:

$$tanh(x) = 2\ S(2x) - 1 \tag{7}$$

[1]Notice how the function $f(x) = ax$, although linear, doesn't include the constant term. This is because the bias of the neuron becomes the constant to the linear activation.

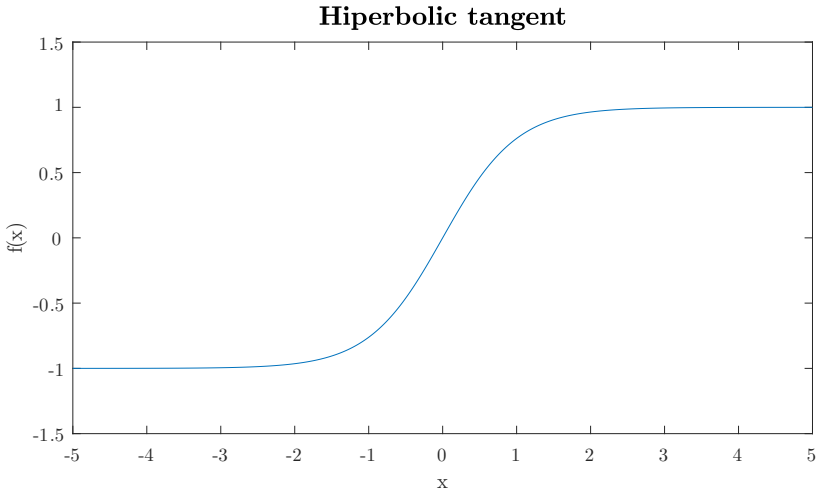


FIG. 3. Tanh funciton

Where $S(x)$ is the sigmoid function. The characteristics of the tanh function are similar to the ones of the sigmoid function. The two main difference are: the bound range of tanh is $(-1, 1)$ instead of $(0, 1)$, the curve of tanh is steeper than the curve of the sigmoid function. This small differences are crucial when trying to optimise the learning process of the network. Again, the *Gradient Descent* algorithm uses the derivative of the activation functions to calculate the change that has to be applied to weights and biases when learning. The derivative of tanh presents a steeper curve and thus, it provides stronger gradients when learning.

2.4. Rectified linear unit (ReLU) $[f(x) = max(0, x)]$**.** ReLU is one of the most used activation functions nowadays. At first glance, it might seem that ReLU presents the same problems as linear functions because it is linear in the positive axis (see Fig. 4). Overall, though, ReLU is nonlinear, and any combinations of ReLU is also nonlinear. This means that ReLU can be stacked in layers, but it is not bounded. Its range goes from 0 to $+\infty$, which means it can blow up the activation of neurons. But ReLU presents a very good advantage compared to the other activations seen so far: sparsity.

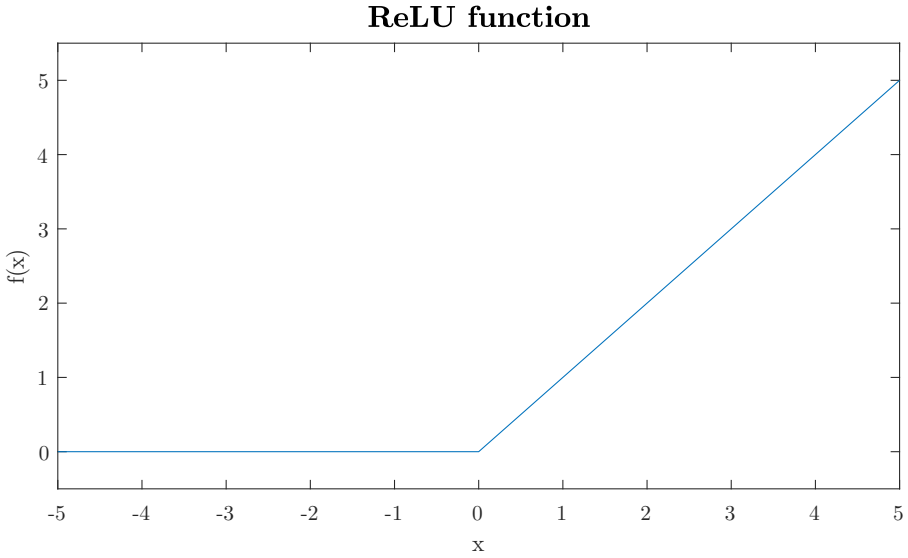


FIG. 4. Relu function

When using an activation function like sigmoid, all neurons in a network will contribute to the final output by producing a real value between 0 and 1. This is computationally very costly, and it would be ideal if only some of the neurons were used every time. ReLU gives this benefit. Because of the straight line at 0 in the negative axis, whenever the weighted sum is negative, the neuron won't activate, which facilitates the calculations for future neurons.

But ReLU also presents a major downside. While learning and adjusting weights and biases, the learning algorithm may favor a neuron to output always a 0 (i.e. the gradient can go towards 0). When this happens, weights and bias for that neuron will not change, which may cause that some parts of the network become passive. To overcome this problem, there are variations of the ReLU activation function. For example, the **Leaky ReLU** (Fig. 5) adds an extra parameter (α) to the neuron that can also be learned. This parameter forces the ReLU to have a very small inclination towards the negative values, which solves the passivity problem.

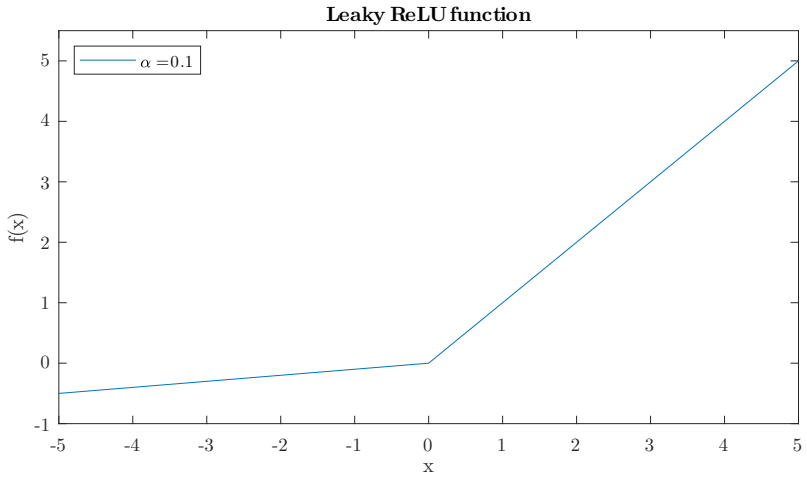


FIG. 5. Leaky Relu function with $\alpha = 0.1$

These are some of the mostly used activation functions right now, but there are plenty of them that are continuously being developed and studied. The type of activation function to use depends a lot on various aspects, such as computational power, network size and structure, output representation, etc.

3. Dropout layers

Dropout layers are sometimes used in conjunction with normal "dense" layers. Dense layers is the name given to neural network layers that fully connect with its prior layer.

Dropout layers are also fully connected with its previous layers, but at some points during the training process the neurons in this layers may randomly deactivate for a single training sample. Not all at once, but just a few of them.

They are used as a regularization technique to reduce overfitting of the network with the training dataset.

4. The output layer

As previously mentioned, the output layer can have one or more neurons and, once the network has trained, it will give the solution to the initial problem. Thus, the representation of the data for these final neurons is very important and it requires special attention when designing the network. Depending on whether the initial problem is a regression one or a classification one, the design of the output layer might be different.

Although it is possible to solve regression problems using Artificial Neural Networks, through a method known as General Regression Neural Networks (GRNN) [**8**],

neural networks tend to perform better with classification problems. Since this project uses classification in its practical part, the theoretical part will also focus on how to approach a neural network that deals with classification. Depending on which type of classification the network is working on, the output layer must be devised in a different manner.

4.1. Logistic regression (binary classification). Although logistic regression is a form of regression, its usually taken to apply to a binary dependent variable, which is estimated continuous or categorical parameters. Since the output of a logistic regression is binary, a single neuron can be used in the output layer. This neuron will preferably have an activation function with fixed bounds, like sigmoid ($[0, 1]$) or tanh ($[-1, 1]$). This will help to classify an input to the network: if the final neuron outputs a value > 0.5 (or 0 in case of tanh), we classify the input as A, B otherwise. Even the threshold (or step) function can be used when dealing with binary classification.

4.2. Multi-class classification. When dealing with multi-class classification, using a single neuron for the output layer can be cumbersome. A common technique used to carry out multi-class classification is to reduce it to multiple binary classification problems. This strategy is also called *One-vs-all* technique [**9**]. Instead of having one neuron to predict N classes, the output layer consists of N neurons, each one specialised in one class. In other words, a single binary classifier (or neuron) is used for every class, and it is trained to only detect elements of that class. For example, training a network to recognise single handwritten digits will have 10 output neurons, one for every possible number (0...9). When a new digit is fed to the network, each output neuron will give a real value between 0 and 1 (if working with sigmoid activation functions), which can be interpreted as the confidence of the network for that digit to be each number. The final output of the network will then be the maximum value of all neurons:

$$\hat{y} = \underset{k \in \{1 \dots K\}}{\operatorname{argmax}} f_k(x) \tag{8}$$

The problem of using argmax is that the end result of all output neurons is not a probability distribution. In binary classification or simple logistic regression, the output neuron gives us a probability distribution among two classes. If the output of such neuron is 0.84, this means that the probability of class A is 0.84, whereas the probability of class B is $1 - 0.84 = 0.16$. In contrast, when doing multi-class classification with N neurons, the interpretation of each value is different. If the third neuron outputs a value of 0.73, the interpretation would be: the probability of class C (third neuron) is 0.73, whereas the probability of any other class is 0.27.

To overcome this problem, the softmax transformation is used. Softmax will assign decimal probabilities to each class for a multi-class classification. The sum of all probabilities must add up to 1.0. This additional constrain will help training converge more quickly. The Softmax function is given by:

$$f(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \tag{9}$$

Where K is the number of classes. A simple example will help understand the concept of multi-class classification. Let us say we are trying to classify an image of a fruit. We know that all images fed to the network will be one of the following types: Banana, Apple, Pear or Melon. This means that the initial problem is a multi-class classification problem with 4 possible classes. It's important to notice that in each image there will be one and only one type of fruit.

Because we are working with 4 classes, the output layer will have 4 neurons with sigmoid activation functions. Once the network is trained and fed with an image of a banana, the neurons may output something like:

- First neuron (banana): 0.95
- Second neuron (apple): 0.48
- Third neuron (pear): 0.21
- Fourth neuron (melon): 0.61

These values can be understood as follows:

The first neuron (banana) is 95% sure that the image is a banana. The second neuron (apple) is 48% sure that the image is an apple. The same goes for the third and fourth neurons. Once we apply the softmax function, the values given are:

- First neuron (banana): 0.36
- Second neuron (apple): 0.22
- Third neuron (pear): 0.17
- Fourth neuron (melon): 0.25

This time, the neurons are not telling what they think separately, but all together. This is the probability distribution of the image just fed. In this case, because the first neuron outputs the greatest value, we would classify the image as a banana.

One could argue that using Softmax is not necessary, and simply applying standard normalization to the values would not only work but also, and more importantly, decrease computational cost. While its true that the softmax function is costly computationally speaking, it would not work as efficient as softmax.

4.3. Muti-label multi-class classification. Dealing with multi-label multi-class classification means that the neural network has to handle different classes, with different labels for each class. Following the example introduced before, imagine now that apart from fruits, there is also an animal in the image. That is, a fruit and an animal in the same image. What before had been classes of fruit are now labels for the class fruit (banana, pear, apple and melon). The same applies for the class animals. In such case, the output layer would have $a + b$ neurons where a are the number of labels for class A and b are the number of labels for the class B. Now, using Softmax will not help to predict which animal and fruit are there in the image because Softmax will treat all neurons under the same probability distribution. A

plausible solution would be to apply N Softmax functions, where N stands for the number of classes involved. This will introduce a lot of problems during the learning process, making it very slow. The more simple and efficient solution is to use a logistic regression for each neuron and then select which fruit and which animal applying argmax to each class. *Sidenote: this type of classification is the one used in this thesis, and will be further discussed in the practical framework.*

5. The cost function

The cost function is a measure of how wrong the network is in terms of its ability to estimate the relationship between the input data (X) and the target output (y). This is normally expressed as a difference or distance between the predicted value and the actual value. This cost function will help the network in the process of learning. The objective of the neural network, therefore, is to find parameters (weights and bias) that minimize the cost function.

Following the example presented in the previous section, we now have to design a cost function that will tell the network how far away its outputs are from the actual values. Any form of formula that computes distances between two points in a parametric space is valid as a cost function. Although this is true, the more commonly used function is the euclidean distance. The Euclidean distance states that the distance between $\mathbf{p} = (p_1, p_2, ..., p_n)$ and $\mathbf{q} = (q_1, q_2, ..., q_n)$ is calculated as follows:

$$d(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{n} (q_i - p_i)^2 \tag{10}$$

Notice that no square root is used in this equation, despite the Euclidean distance having it. In optimization tasks like this, the square root is not calculated because it increases monotonously and its derivative is very difficult to calculate.

The predicted value corresponds to the vector of probabilities calculated using the softmax function from the output neurons. The actual value is represented as a *one-hot vector*. A one-hot vector is a vector of size N, where N is the number of possible classes, filled with 0's anzd a single 1. The single 1 tells what class the vector is representing. In the previous example with fruits, a one-hot vector representing *banana* would be: $[1, 0, 0, 0]$. This is the actual value that we expect the network will give us. Calculating the Euclidean distance between the predicted vector and the actual vector gives us a value that tells how well the network is doing on predicting fruits.

The goal of the learning process is to minimize this value for any inputted image given to the network. To accomplish this, the learning algorithm *Gradient Descent* is used.

It's important to distinguish the cost function from the overall Neural Network function. The neural network function can be represented as follows:

$$f(X_n; \Theta) \rightarrow Y_m \tag{11}$$

Where X is the input vector to the network, n is the size of the vector (i.e. the size of the input layer) and m is the size of the output vector. The parameters of the Neural Network function are all the weights, biases and other parameters (like the one introduced by Leaky ReLU), which constitute the parametric space Θ. The output of the neural netowrk is a vector Y, where m is the size of the vector (i.e. the size of the output layer).

The cost function can be represented as follows:

$$f(\Theta; \textit{training example}) \rightarrow \textit{error} \tag{12}$$

Where Θ is the parametric space and *error* is the error produced by the cost function. The cost function is calculated for every training example to see how well the network performed and tune the parametric space to minimize the cost function using the Gradient Descent algorithm.

6. The learning algorithm. Gradient descent

Up until now, we know that the process of learning (or training) a neural network involves tuning its weights and biases (the parametric space) so that the cost function is the lowest possible. For convenience, from now on, the parametric space will be called Θ, and will include all parameters that define a neural network. θ defines a point in the parametric space Θ. The cost function will be called C.

If Θ consisted only of one variable, finding the points at which the derivative of C equals 0 would give us the maxima and minima of the cost function. With the second derivative, the minimums of C could be found.

$$\frac{dC(\theta)}{d\theta} = 0 \qquad \frac{d^2C(\theta)}{d\theta^2} > 0 \tag{13}$$

There's a problem when using derivatives with neural networks. The number of parameters to tune in a neural network grows exponentially with respect to the number of neurons and number of layers. This makes the derivative approach impractical.

Another approach is to randomly generate the values of Θ and then figure out which direction to go in order to decrease the output of C. Specifically, by figuring out the slope of C at the point of Θ, the input can be shifted to the direction where the slope decreases. In multivariate calculus, the *Gradient* of a function gives the direction of steepest ascent given a point.

$$\nabla C(\theta) \tag{14}$$

In the case of the Cost function C, the *Gradient* will give the direction vector to which the value of C increases most quickly. By inverting this value, the direction vector now points towards the steepest descent will be obtained. Thus, the *Gradient descent* can be defined as the negative of the *Gradient* of the cost function at point θ:

$$-\nabla C(\theta) \tag{15}$$

By doing this repeatedly at each training step, a minimum will be eventually found. This minimum will most likely be a local minimum. Trying to find the global minimum using this method is extremely complicated and requires a lot of repetitions starting w randomly at each repetition. This is the downside of using *Gradient Descent* but is still more efficient and accurate than trying to derivate the cost function.

As just mentioned, the returned value of $-\nabla C(\theta)$ will be the direction vector to which the output of C decreases more quickly. Thus, by adding this vector to the current value of θ, we will be a step closer to a minimum of C.

$$\theta_n = \theta_{n-1} + (-\nabla C(\theta_{n-1})) = \theta_{n-1} - \nabla C(\theta_{n-1}) \tag{16}$$

6.1. Learning rate. There's another downside with respect to *Gradient Descent*. Once the *Gradient Descent* has been calculated, the changes needed to apply to the parameters might be so big that they may overshoot the minimum of the cost function and make the loss worse. To overcome this problem, a new variable is introduced: the *learning rate* (η). This parameter tells the optimizer (the *Gradient Descent*) how far to move the weights in the direction opposite of the gradient.

$$\theta = \theta - \eta \nabla C(\theta) \tag{17}$$

Again, if the learning rate is big, the optimizer may overshoot. On the contrary, if the learning rate is very low, the training is more reliable but optimization will take a lot of time because steps towards the minimum of the loss function are tiny. This concept is represented in Fig. 6.

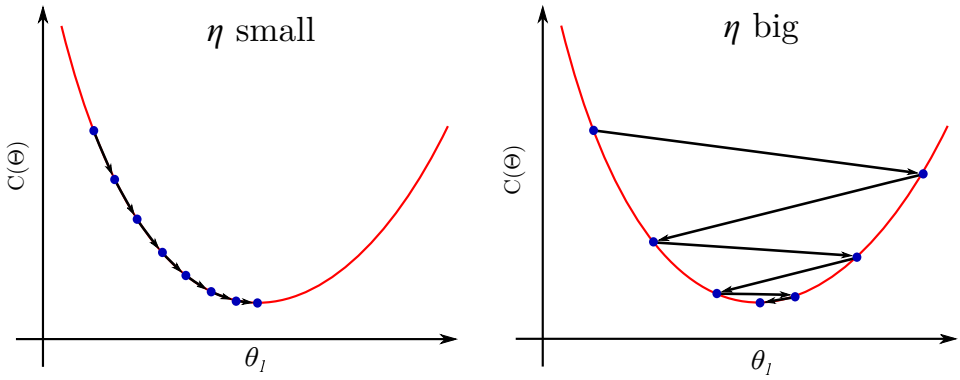


FIG. 6. This figure shows the difference between a small learning rate and a large learning rate. The parametric space Θ represented here consists only of one parameter θ_1. When the learning rate is small, there are more steps and smaller, which means more computational power needed. When the learning rate is big, less steps are needed, but the optimizer may overshoot.

The training should start with a relatively large learning rate because in the beginning the θ is far from optimal. During the training, the learning rate can decrease to allow more fine-grained weighted updates.

6.2. Stochastic Gradient Descent (Mini-batches). Ideally, the Gradient Descent should be calculated for all the training samples at each step, and the mean vector should be applied to θ. Because this is extremely costly computationally speaking, instead, at each step only a batch of samples is taken into consideration. The mean of the Gradient Descent computed for every training sample in the batch is then applied to θ. This way, the path to the minimum value isn't as direct as it would be, but the computational gain is much greater. This concept is represented in Fig. 7. The size of each batch depends on the total number of training samples and the actual computational power.

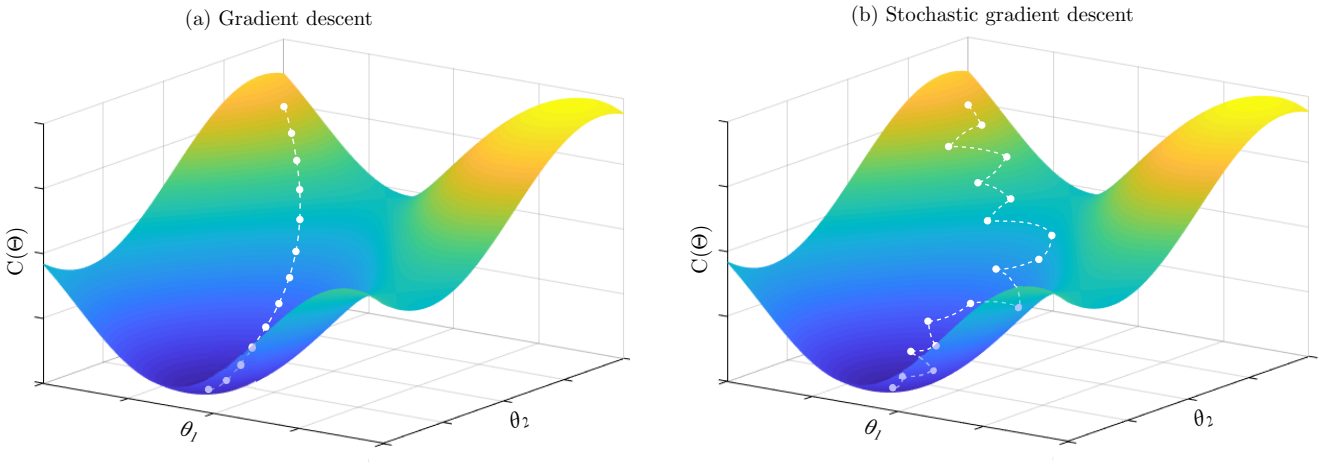


FIG. 7. This figure shows the difference between Gradient Descent and Stochastic Gradient Descent. The parametric space Θ represented here consists of two variables θ_1 and θ_2. In Gradient Descent, there are less steps that go towards the minimum at each step. This is because all the training data is used at each step, which means a lot of computational power is needed. In Stochastic Gradient Descent, only a mini-batch of training samples is used which makes the optimizer go in different suboptimal directions, but always towards the minimum. Stochastic Gradient Descent requires more steps with less computational power.

7. Backpropagation calculus

In this last section about Artificial Neural Networks, a more in depth look will be given to the calculus behind the learning process. In the previous sections, a general idea was given about Gradient Descent, Cost function and how both are used to tune the parameters of the network.

Backpropagation is the name given to the process of changing the weights and biases of the network. It is called backpropagation because the process consists on changing first the last layer parameters and going all the way to the input layer in reverse order. To illustrate how the process of backpropagation works, a very simple neural network will be introduced showed at Fig. 8.

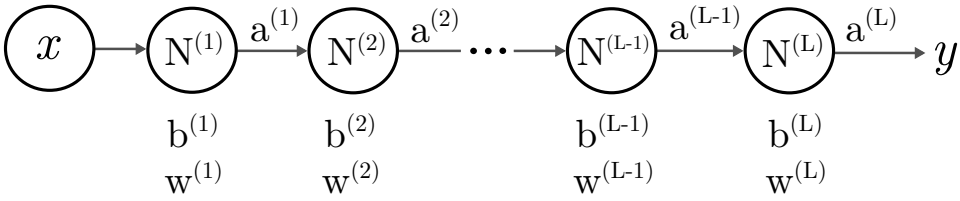


FIG. 8. The network used as an example for this section consists of L neurons one after the other. IThere is only one neuron per layer. $N^{(i)}$ is the ith neuron. $w^{(i)}$ and $bb^{(i)}$ are the weight and bias of $N^{(i)}$. The output of each neuron is represented by $a^{(i)}$.

This simple neural network consists of $L-1$ hidden layers with one and only one neuron per layer. Following the same pattern, the output layer consists of only one neuron. This means that the overall parametric space of this network is determined by L weights and L biases:

$$\theta = \{w^{(1)}, b^{(1)}, w^{(2)}, b^{(2)}, ..., w^{(L-1)}, b^{(L-1)}, w^{(L)}, b^{(L)}\} \tag{18}$$

The cost function of such neural network is:

$$C(w^{(1)}, b^{(1)}, w^{(2)}, b^{(2)}, ..., w^{(L-1)}, b^{(L-1)}, w^{(L)}, b^{(L)}) \tag{19}$$

For each neuron, the terminology used to represent its output value will be $a^{(i)}$, where a stands for *activation* of the ith neuron. This means that the cost of this network for the first training example in a mini-batch (C_0) can be simplified as follows:

$$C_0(...) = (a^{(L)} - T)^2 \tag{20}$$

where T is the expected output for the training example 0, and $a^{(L)}$ is computed using the output of the previous neuron ($a^{(L-1)}$):

$$a^{(L)} = f(a^{(L-1)}w^{(L)} + b^{(L)}) \tag{21}$$

Where f is the activation function of the last neuron (sigmoid, ReLU, etc). For further reference, $y^{(L)} \equiv a^{(L-1)}w^{(L)} + b^{(L)}$, which means that the output of the last neuron will be:

$$T = a^{(L)} = f(y^{(L)}) \tag{22}$$

As mentioned in the previous section, the main goal of *Gradient Descent* is to minimize the Cost function step by step. In other words, what *Gradient Descent* does is to find how sensitive the cost function is to changes to the weights and biases. If the Cost function is very sensitive to a specific weight $w^{(k)}$, the value given by $-\nabla C$ for $w^{(k)}$ will be very high in **absolute terms** *(remember that the Gradient Descent tells to which direction Θ should go. The sensitivity of the cost function for w_k might be very big but towards the negative axis, i.e. negative in value).* If, instead, the cost function is not sensitive to a specific bias $b^{(k)}$, the value given by $-\nabla C$ for $b^{(k)}$ will be very low.

The sensitivity of the cost function to a particular parameter (weight or bias) is the derivative of the cost function with respect to that particular parameter. This is what the *Gradient* calculates for each parameter. In our neural network example, the *Gradient Descent* will be:

$$-\nabla C(w^{(1)}, b^{(1)}, ..., w^{(L)}, b^{(L)}) = -\begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \frac{\partial C}{\partial w^{(2)}} \\ \frac{\partial C}{\partial b^{(2)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L-1)}} \\ \frac{\partial C}{\partial b^{(L-1)}} \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix} \tag{23}$$

So far, the following equations have been introduced:

$$\begin{gathered} y^{(L)} = a^{(L-1)}w^{(L)} + b^{(L)} \\ a^{(L)} = f(y^{(L)}) \\ \Downarrow \\ C_0(...) = (a^{(L)} - T)^2 = (f(a^{(L-1)}w^{(L)} + b^{(L)}) - T)^2 \end{gathered} \tag{24}$$

Because the cost function depends directly on $w^{(L)}$, $b^{(L)}$ and $a^{(L-1)}$, these will be the first derivatives to be calculated by the *Gradient Descent*. Starting with $w^{(L)}$ and using the equations above:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial y^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial y^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}} = a^{(L-1)} \cdot f'(y^{(L)}) \cdot 2(a^{(L)} - T) \tag{25}$$

This is called the *Chain rule*. Multiplying these three ratios gives the sensitivity of C_0 to small changes of $w^{(L)}$ for training example 0. As explained in the previous section, during training, mini-batches are used to decrease computational cost. After each mini-batch, the mean value of the n training examples in the mini-batch will be calculated:

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{(k=0)}^{(n-1)} \frac{\partial C_k}{\partial w^{(L)}} \tag{26}$$

This resulting value $\frac{\partial C}{\partial w^{(L)}}$ will be the one given by ∇C for the parameter $w^{(L)}$. In other words, the value given is the change needed to apply to the current value of the parameter $w^{(L)}$ in order for the Cost function to increase towards the steepest slope. Because we want the cost function to decrease, the negative value will be given by $-\nabla C$.

The *Chain Rule* can then be applied to $b^{(L)}$:

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial y^{(L)}}{\partial b^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial y^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}} = 1 \cdot f'(y^{(L)}) \cdot 2(a^{(L)} - T) \tag{27}$$

Again, the mean value will be obtained for all training examples in the mini-batch:

$$\frac{\partial C}{\partial b^{(L)}} = \frac{1}{n} \sum_{(k=0)}^{(n-1)} \frac{\partial C_k}{\partial b^{(L)}} \tag{28}$$

Finally, and here is were the idea of backpropagation comes in, the *Chain Rule* will be applied to $a^{(L-1)}$.

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial y^{(L)}}{\partial a^{(L-1)}} \cdot \frac{\partial a^{(L)}}{\partial y^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}} = \boldsymbol{w^{(L)} \cdot f'(y^{(L)}) \cdot 2(a^{(L)} - T)} \tag{29}$$

This is the sensitivity of C to changes of the output value given by the previous neuron. Because the output value of the previous neuron can't be directly modified (in other words, $a^{(L-1)}$ is not a parameter), we will have to look backwards and modify the bias and weights of the previous neuron instead. But the equation given $\boldsymbol{w^{(L)} \cdot f'(y^{(L)}) \cdot 2(a^{(L)} - T)}$ will be used to compute the derivative of the cost

function with respect to previous biases and weights. This means following the *Chain Rule*. The derivative of the cost function with respect to the weight from the second to last neuron ($w^{(L-1)}$) is shown as an example:

$$\frac{\partial C_0}{\partial w^{(L-1)}} = \frac{\partial y^{(L-1)}}{\partial w^{(L-1)}} \cdot \frac{\partial a^{(L-1)}}{\partial y^{(L-1)}} \cdot \frac{\partial y^{(L)}}{\partial a^{(L-1)}} \cdot \frac{\partial a^{(L)}}{\partial y^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}} = \tag{30}$$

$$= a^{(L-2)} \cdot f'(y^{(L-1)}) \cdot \boldsymbol{w^{(L)} \cdot f'(y^{(L)}) \cdot 2(a^{(L)} - T)}$$

The same will apply for $b^{(L-1)}$ and all parameters from all previous neurons, completing the *Chain Rule*.

The calculus done so far applies only to our example because each layer has one and only one neuron. Once this first example is understood, the generalization of this idea with multiple neurons at each layer is very straightforward.

From now on, two subindexes to each weight will be added that represent the connection between a neuron in the previous layer and a neuron in the actual layer. In other words, weight $w_{jk}^{(L)}$ is the weight of the jth neuron in the last layer (L) that connects to the kth neuron of the previous layer. The activations and biases will have only one subindex, so that $b_j^{(L-1)}$ would be the bias of the jth neuron from the second to last layer and $a_j^{(2)}$ would be the output of the jth neuron from the second layer. This generalized neural network is represented at Fig. 9.

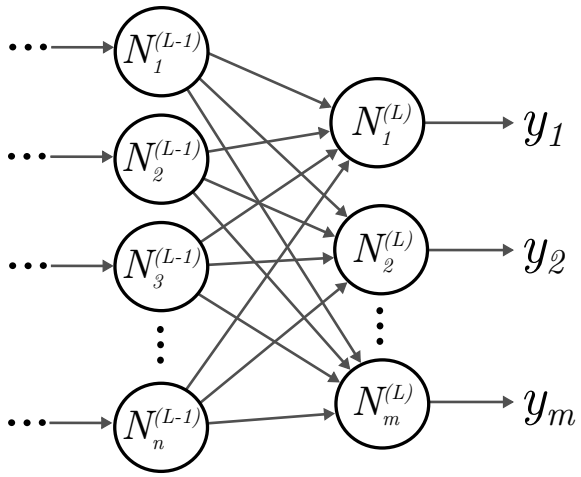


FIG. 9. This figure shows the terminology used by real neural network (with more that one neuron per layer). Only the last two layers are presented. $N_j^{(i)}$ represents the jth neuron in the ith layer. n and m are the number of neurons in the second to last and last layers respectively. The weight between $N_j^{(i)}$ and $N_k^{(i+1)}$ is represented as $w_{kj}^{(i+1)}$

In this more generalized example, the cost function for the first training example of a mini-batch will be (using Euclidean distance as presented before):

$$C_0 = \sum_{j=0}^{n_L-1} (a_j^{(L)} - T_j)^2 \tag{31}$$

Where n_L is the number of neurons in the last layer. This same equation was presented at [equation 10] when introducing the cost function. Again, the relative weighted sum for the jth neuron in the output layer will be called $y_j^{(L)}$:

$$y_j^{(L)} = a_0^{(L-1)} w_{j0}^{(L)} + a_1^{(L-1)} w_{j1}^{(L)} + ... + a_{n_{(L-1)}}^{(L-1)} w_{jn_{(L-1)}}^{(L)} + b_j^{(L)} \tag{32}$$

Where $n_{(L-1)}$ is the number of neurons in the second to last layer. The output of the jth neuron in the last layer can now be represented as:

$$T_j = a_j^{(L)} = f(z_j^{(L)}) \tag{33}$$

Now, the derivative of $w_{jk}^{(L)}$ and $b_j^{(L)}$ with reference to C are calculated as follows:

$$\begin{aligned} \frac{\partial C_0}{\partial w_{jk}^{(L)}} &= \frac{\partial y_j^{(L)}}{\partial w_{jk}^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial y_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}} \\ \frac{\partial C_0}{\partial b_j^{(L)}} &= \frac{\partial y_j^{(L)}}{\partial b_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial y_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}} \end{aligned} \tag{34}$$

Up until now, little has changed with respect to the first example, despite the derivatives of each parameter that will be different. The difference comes when trying to compute the derivative of the cost function with respect to the previous output values. Since now there are multiple neurons in the previous layer, each one of them influences all of the neurons in the last layer. In other words, the output of the jth neuron in the last layer is influenced by $a_k^{(L-1)} \; \forall \; k \; \in \{1, ..., n_{(L-1)}\}$ where $n_{(L-1)}$ is the number of neurons in the second to last layer. This means that the sum of all the derivatives with respect to each neuron in the previous layer must be calculated:

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial y_j^{(L)}}{\partial a_k^{(L-1)}} \cdot \frac{\partial a_j^{(L)}}{\partial y_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}} \tag{35}$$

This can be understood as: the sensitivity of C_0 to small changes in the output of the kth neuron in the second to last layer ($a_k^{(L-1)}$) is the sum of all small influences of the kth neuron from the second to last layer to each and every one of the neurons

in the last layer. Again, the output if the kth neuron from the second to last layer can't be modified because it is not a parameter, but this derivative will help compute further derivatives going back through the *Chain Rule*.

In the end, the mean value taken among the derivatives of all training examples in a single mini-batch for every parameter will produce the *Gradient* of the cost function. Inverting the direction of that vector will give us the direction and magnitude to which modify each parameter in our parametric space Θ in order to decrease the cost function.

8. Brief recap

The main goal of any neural network is to minimize the cost function. Normally, this cost function will be defined by the Euclidean distance.

The only values that can be tuned in the neural network are the biases and weights, which conform the parametric space Θ. Knowing this, the optimal values must be found so that the cost function minimizes. The *Gradient Descent* algorithm will apply small nudges to the weights and biases at each step (consisting of a mini-batch) towards a local minimum of the cost function.

In the last chapter, the process of how *Gradient Descent* caluclates the small nudges needed to apply to each parameter was deeply revised. During this process, the *Gradient Descent* calculates the derivative of the activation function of every neuron. Knowing this, it now makes sense why some activation functions are better than others.

If a linear function is used as an activation function for a neuron, the derivative will be a constant, which means that the weights and biases for that neuron will always be tuned with a constant by *Gradient Descent*. Furthermore, the input value to the neuron will not matter and won't contribute to the *Gradient Descent*, which will make the learning process impossible.

Comparing the sigmoid function and the tanh function, it now makes sense why tanh function might be better in some occasions. The derivative of the tanh function a greater slope which translates to bigger gradients towards a minimum of the cost function.

Overall, this chapter has introduced Artificial Neural Networks and the in-depths of its workings. In the next chapter, Convolutional Neural Networks will be discussed, which are a more recent type of neural networks, but are based upon the same concepts and ideas introduced in this chapter.

Chapter 4
Covolutional Neural Networks

1. Deep Learning

In this chapter, an overview will be given about more complex models based on the basic neural networks presented in the previous chapter.

As explained earlier, single neurons can be merged together to form what are called neural networks. In this neural networks, the output of a neuron becomes input for other neurons, always following a feed-forward pattern (i.e. not going backwards at any moment). In fact, there is a type of neural networks that allow this kind of connections: the ouptut of a neuron can be the input of a previous neuron in a previous layer. This type of neural networks are called Recurrent Neural Networks [**10**], with variants like Long-Short Term Memory Neural Networks (LSTM) [**11**]. This project, though, will focus only on feed-forward neural networks.

Before continuing, it's important to explain the difference between shallow neural networks and deep neural networks. A shallow neural network is a neural network with one or two hidden layers (without counting input and ouptut layers). Deep neural networks, in contrast, can have more than two hidden layers. Nowadays, deep neural networks are widely used everywhere.

In 1989, Cybenko proved the **Universal Approximation Theorem** [**12**], which states that a feed-forward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous real function on a bounded interval. However, the proof of this theorem is not constructive, and therefore does not offer an efficient training algorithm for learning such structures in general.

There are 4 possible answers to why shallow neural networks don't learn as quickly and efficiently as deep neural networks, although the Universal Approximation Theorem states the contrary:

(1) A shallow network maybe needs more neurons then the deep one.
(2) A shallow network is more difficult to train with our current algorithms (e.g. it has more nasty local minima, or the convergence rate is slower).

(3) A shallow architecture does not fit to the kind of problems we are usually trying to solve.

In the *Deep Learning* book [**13**] strong emphasis is given to bullet points #1 and #3. First, it argues that the number of units in a shallow network grows exponentially with task complexity. This means that a shallow neural network should be much more bigger in *width* than the deep neural network in *depth*. Secondly, and quoting the book:

"Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation."

Also, deep neural networks provides an advantage against shallow neural networks. The advantage is that deep learning in general provides a way of learning data representation, as opposed to task-specific algorithms provided by shallow neural networks. In other words, the neural network can automatically discover the representations needed for feature detection or classification from raw data. This replaces manual feature selection and allows a machine to both learn the features (unsupervised learning) and use them to perform a specific task (supervised learning).

Usually, with a supervised neural network, the vector y is being predicted from an input vector x. But when training an unsupervised neural network, the x vector becomes the input and the target variable at the same time. Doing this, the network can learn something intrinsic about the data without the help of a target or label vector that is often created by humans. This type of structure leads to the hourglass shape that is common when training unsupervised, deep neural networks (see Fig. 1). In such a shape, each hidden layer becomes a lower dimension projection of the layer before it as well.

Once the unsupervised learning is completed, a supervised task-specific learning algorithm can be applied. For that, the network is typically split in half and the output vector is replaced with y. Now, the starting point for this supervised learning algorithm is the end point of the unsupervised learning process. When doing this, the weights and biases previously trained using unsupervised methods are now the initial point for the supervised task.

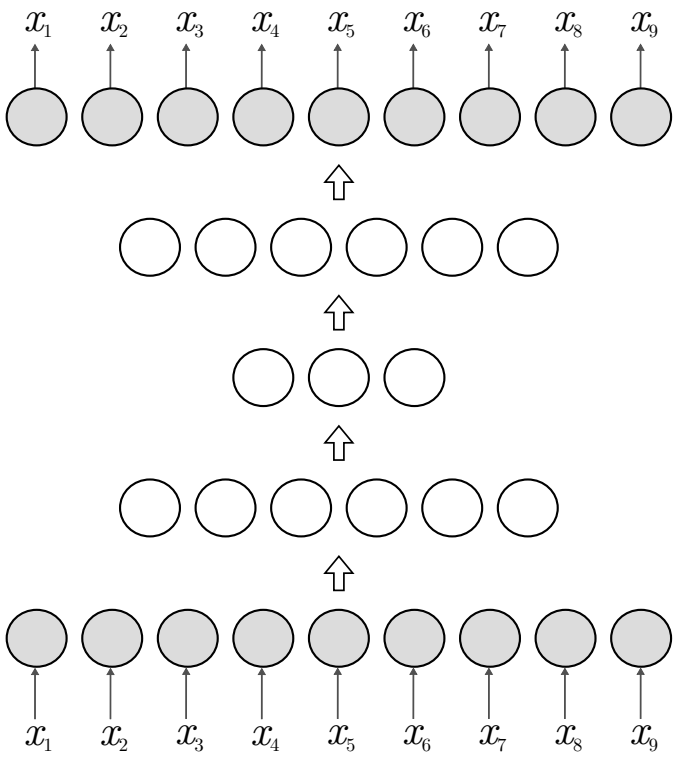


FIG. 1. This process shows the hourglass shape network used in unsupervised learning. The input vector and the target vector is the same. This means that the network needs to find the important features that define the input vector in order to reconstruct it as an output.

2. Convolutional Neural Networks

Convolutional Neural Networks are a special type of Deep Neural Networks that are very popular in image recognition, video analysis, natural language processing, drug discovery and board games among other areas of research.

Convolutional Neural Networks takes the basis of Artificial Neural Networks (fully-connected neural network, like the ones presented in the previous chapters) and it adds a level of complexity. In fact, a whole Convolutional Neural Network can be split in two structures: a group of layers specialized in pattern recognition and a fully-connected network. The fully-connected layer is just an Artificial Neural Network appended to the first part. The term *fully-connected* neural network emphasizes the fact that all neurons at layer N are connected to all neurons at layer $N + 1$.

As briefly explained earlier, Convolutional Neural Networks are widely used in image recognition and image classification. This type of neural networks are currently used by huge and popular companies like Facebook, Google, Pinterest, Amazon and

others. Because understanding Convolutional Neural Networks is easier when dealing with image recognition and classification, this introduction will first expound how Convolutional Neural Networks deals with images. Later on, the same concept will be applied to board games, like Chess or Go.

Image classification is the task of taking an image as the input and outputting a probability distribution of classes (eg. an apple, a banana, a pear or a melon). This type of task is learned by humans in the early stages of childhood, and is one that comes naturally and effortlessly as adults. Without hesitation, we can quickly and seamlessly identify objects in our environment. When looking at an image or our surroundings, we are able to label each object. This skills are the ones that need to be taught to the machines in order for them to classify images as we can.

But why aren't simple Multi-layered Perceptrons used for such problems? Why do we need Convolutional Neural Networks? The reason is that such a network architecture does not take into account the spatial structure of the images. For instance, it treats input pixels which are far apart and close together on exactly the same manner. Such concepts of spatial structure must instead be inferred from the training data. Convolutional Neural Networks use a special architecture which is particularly well adapted to classify images.

While humans perceive images through light, computers don't. Instead, computers understand images as a matrix of $X \times Y \times 3$ real values, where X and Y are the length and width of the image, and 3 corresponds to the Red-Green-Blue (RGB) color representation of a pixel. The goal for the computer is to take an $X \times Y \times 3$ matrix as input and output a probability distribution class. To achieve this, we want the computer to figure out the unique features that make a banana a banana, an apple an apple or a melon a melon. This is the process that goes on our minds when we classify an object. We detect certain patterns, edges or colors that form a particular object. This process will take place in the first part of a Convolutional Neural Network (the group of layers specialized in pattern recognition, convolutional layers).

Just like Artificial Neural Networks, Convolutional Neural Networks are based upon the structure of the brain. In this case, Hubel and Wiesel in 1962 [**14**] showed that some individual cells in the visual cortex of mice fire when detecting edges with a certain orientation. They also found out that all this cells are organized in a columnar architecture, going through more complex pattern recognition neuron layers each time.

The explanation will start with simple black and white images of size 32×32 pixels. This reduces the depth of an image from 3 dimensions (3 colors, RGB) to just 1, a number between 0 and 255 representing a color in the gray scale. This will help understand how convolutional layers work.

Convolutional Neural Networks use three basic ideas: *local receptive fields*, *shared weights and bias*, and *pooling layers*.

2.1. Local receptive field. When the MLP was introduced, the input layer was depicted as a vertical vector of neurons. In convolutional neural networks, it's best

to imagine the input layer as a matrix of 32×32 neurons. The values of this neurons are the color representation of each pixel in the gray scale. An image of the input layer is represented at Fig. 2.

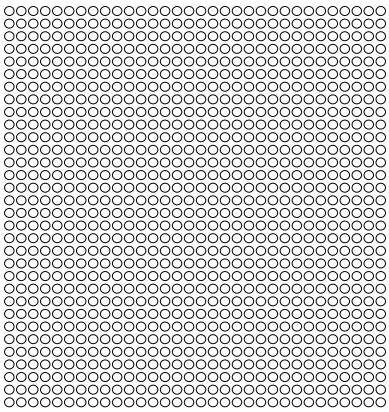


FIG. 2. The input layer of a CNN is constructed as a matrix of neurons rather than a vector. This helps keeping the spatial relation between neurons (i.e pixels of the image)

This input layer will be connected to the first hidden layer, but not in the same way as it was in MLP. Instead, each neuron in the first hidden layer will be connected only to a small set of neurons in the input layer. This way, each neuron in the first hidden layer will focus only on a region of the input image. This small area is typically known as *receptive field* and can be of any size. In this example, it will be of size 5×5 pixels.

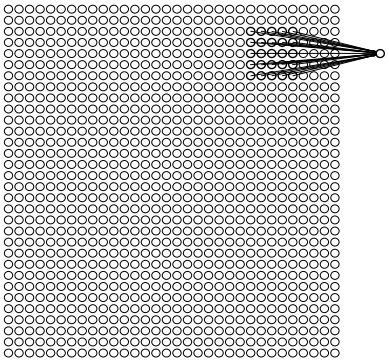


FIG. 3. The receptive field of a single kernel is represented in this figure. The neuron that this receptive field corresponds to is represented on the right of the input layer.

The region highlighted in the first hidden layer is the *receptive field*. Like in MLP, each connection between the 25 neurons in the input layer and the single neuron

in the first hidden layer represents a weight. The matrix of 5×5 weights is called a **kernel**. Like in MLP, each neuron will have a bias.

The kernel will slide across the entire image, starting in the top left all the way to the bottom right corner in single steps of 1 pixel. At each step, a neuron in the next convolutional layer will be specialized in its personal receptive field. This process is sometimes referred as *convolving* an image, and it is shown in the image below. Once a kernel has slid across all image, the output matrix generated by the next hidden layer is called *feature map*.

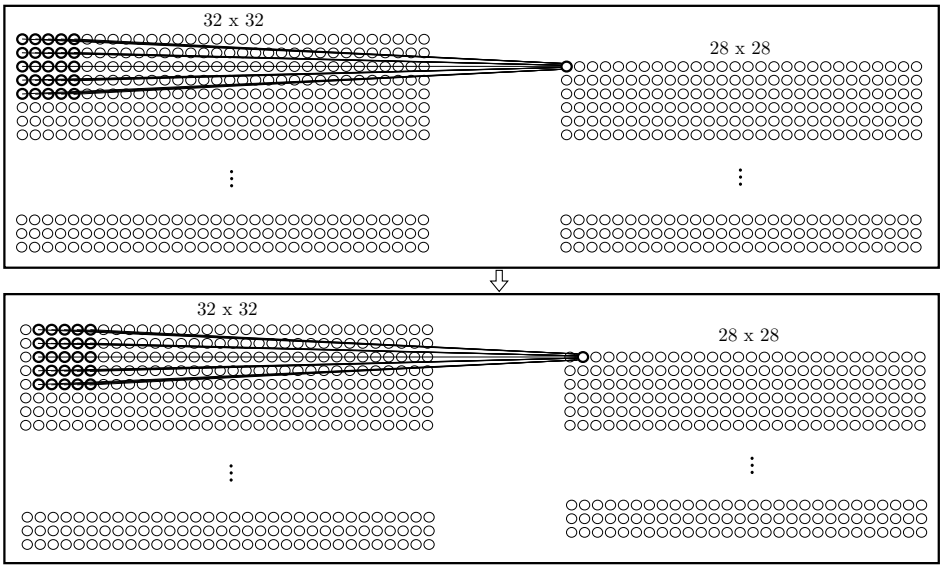


FIG. 4. This figure shows the process of convolving an image. The first square shows the convolution of the first neuron in the first hidden layer. Once the computation has been done, the convolution of the next neuron in the hidden layer is computed (second square). Because the kernel of size 5×5 can only be slid in 28 positions in each direction the feature map of this kernel will be of size 28×28.

It's important to note that having an input image of size 32x32 and a receptive field of size 5x5 will result in a hidden layer of size 28×28. This is because the receptive field will always be filled with input image pixels. Different *padding* techniques can be used. For example a 0 padding of size 3 will make the receptive field start with a margin of 3 pixels outside the image, filled with zeroes.

Another hyperparameter that can be tuned, aside from *padding*, is the *stride*. The kernel can be moved at different "speeds". A *stride* of size 1 will move the kernel 1 pixel at a time, whereas a *stride* of size 5 will move the 5×5 kernel as a whole, generating a feature map with different dimensions.

2.2. Shared weights and bias. As previously mentioned, each neuron in the hidden layer will have its weights and biases, but these will be the same for all 28x28 neurons in the hidden layer. Thus, the activation of the jth, kth neuron will be:

$$y = f\left(\left(\sum_{l=0}^{n}\sum_{m=0}^{n} w_{l,m}x_{j+l,k+m}\right) + b\right) \tag{36}$$

Where $w_{l,m}$ is the weight in the lth, mth position of the kernel and $x_{j+l,k+m}$ is the value given by the input image at offset (l, m) from the first pixel of the receptive field (j, k). The n represents the size of the receptive field (number of neurons at each direction). The weighted sum is then passed through an activation function $f(x)$, that can be any of the previously presented. This computation is normally called *convolution*, and the act of sliding the kernel across the image is the act of *convolving*. This is where the name Convolutional Neural Networks comes from.

This same equation is sometimes referred as follows:

$$y^n = f(b + w * y^{n_1}) \tag{37}$$

Where w and b are the shared weights and bias for that kernel, a^n is the output of the neuron and y^{n-1} is the activation received from the previous layer. The sign operation $*$ is the sign used to represent a *convolution*.

The share of weights and biases by all neurons in the hidden layer means that all neurons in this layer will detect the same pattern, only in different locations of the input image. The feature map generated by the hidden layer is a map indicating at which positions in the input image the pattern represented by the kernel is located at.

The first hidden layer presented so far can only detect one pattern in an image. In fact, various kernels are used at the same time between layers to detect multiple patterns. This means that the feature map of the first hidden layer will consist of $N \times 28 \times 28$ values, where N is the number of kernels used in the hidden layer. This is represented in the following image.

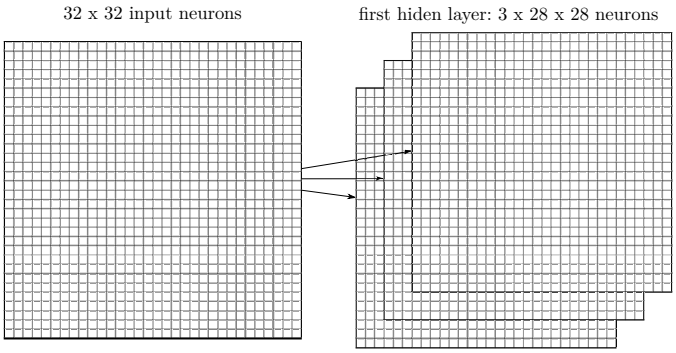


FIG. 5. This figure shows the representation of the first two layers of a CNN. The left of the image shows the input layer of size 32×32 neurons. The first hidden layer is represented at the right. Because this first hidden layer is using three different kernels, the output of the first hidden layer will be $3 \times 28 \times 28$, where 3 is the number of kernels used (or feature maps) and 28 is the width and height of each feature map. This is due to the kernel of size 5×5 that has been used.

One advantage of using shared weights and biases is that it reduces considerably the number of parameters of a CNN.

2.3. Pooling Layers. Convolutional Neural Networks also have what are called pooling layers. This layers are normally put behind every convolutional layer, although they can be put every two or more convolutional layers. Pooling layers are used to simplify the output given by a convolutional layer.

A pooling layer takes each feature map generated by the convolutional layer and condenses it in a more compact image. Given the feature map, for every $m \times m$ square of values, it takes the maximum, where m is the size of the pooling layer. This way, if the pooling size is 2×2 the maximum output of a group of 4 neurons will be preserved. Normally, this technique is called *max-pooling* because it takes the maximum value, but other techniques exist. Another technique could be to get the mean value of the outputs.

By generating a more compact image, one might think that information is lost, which is true, but the information lost is no important. By condensing the feature map, the relative distance between the patterns found in the original image is preserved, but not its exact location. It's not important to know at which point exactly a pattern occurred in an image, just that it occurred and how far away is from other patterns.

Thus, by resizing every feature map, the crucial information of pattern recognition is held while drastically reducing the number of parameters between convolutional layers. An example of a pooling is shown at Fig. 6.

Again, these pooling layers are applied to each feature map returned by the convolutional layer before them. For example, given an output of $6x28x28$ from a convolutional layer (with 6 feature maps), a 2×2 max-pooling layer right after it would output a matrix of size $6x14x14$.

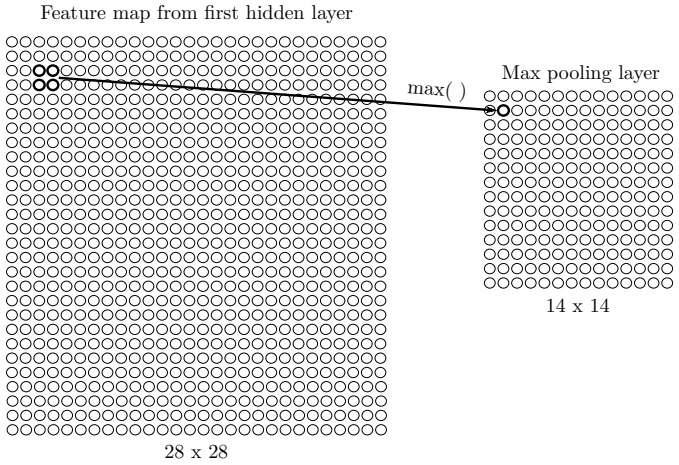


FIG. 6. This image shows the process of max-pooling a feature map. From the four neurons highlighted in the feature map only the one with the highest output will continue the network.

2.4. Overall picture. Now that the concept of a Convolutional Neural Network has been introduced, an overall picture of how a CNN works can be understood.

The first layer of any CNN will always be the input layer containing the image. After that, a convolutional layer will come, with a pooling layer associated with it.

As explained in this introductions, CNN consists of two parts, the pattern recognition part and the fully-connected neural network appended in the end. The role of the convolutional layers (first part) is to detect the important patterns that the image shows, condensing them in smaller but "deeper" (more feature maps) throughout the convolutions. This is because at each convolutional layer, smaller and smaller feature maps are generated. But also, because multiple kernels are applied at each convolutional layer, the overall matrix grows in depth (with more feature maps at each layer).

Once the convolutional part of a CNN is completed, the final result is a $N \times X \times Y$ matrix of neurons, where N is the number of feature maps, X is the length and Y is the width. This output is then fed to a fully-connected neural network.

The fully-connected neural network is appended to perform the task-specific problem, which is the classification task. The first structure of the CNN (the convolutional and pooling layers) are used to detect patterns in the input image. This patterns are then passed through a fully-connected neural network that will try to predict what class those patterns belong to. In other words, it returns a probability

distribution among all possible classes. The following image depicts this structure in more detail.

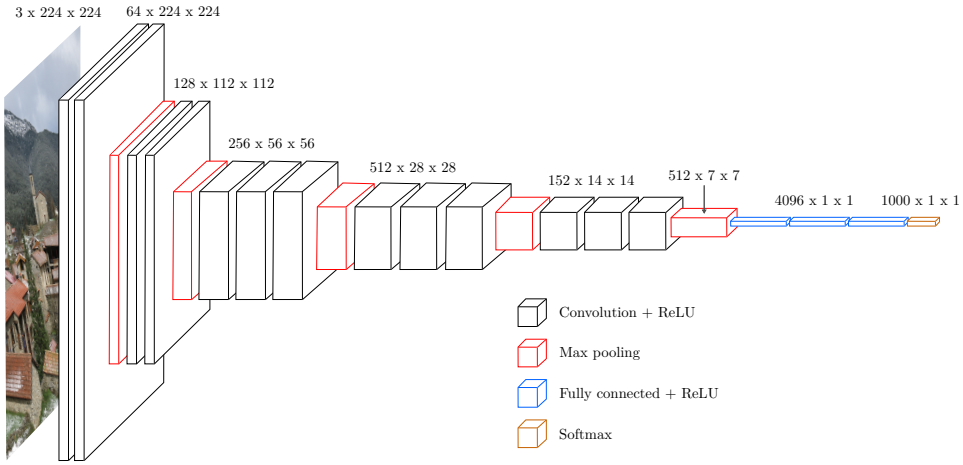


FIG. 7. An example of a full Convolutional Neural Network is represented in this figure. The convolutional layers are colored in black and the max-pooling layers are colored in red. Notice how every two or three convolutional layers there is a max-pooling layer that compresses the dimensions of the feature maps. The final max-pooling layer outputs a matrix of size $512 \times 7 \times 7$ (512 feature maps with 49 values each one), which is then flattened to a 1 dimensional vector before entering the fully-connected network colored in blue. The output of the fully-connected network is filtered by the softmax function and is colored in brown.

3. Training and Backpropagation process in CNN

The process of training convolutional neural networks is the same as the one used when training fully-connected neural networks: *Stochastic Gradient Descent*. The training data is split in mini-batches that are fed together to the network. At each step, the loss function C will decrease towards a local minimum.

The trained parameters of a Convolutional Neural Network are all the parameters present in the convolutional layers plus the ones used in the fully-connected neural network. The Gradient function will again find the direction towards the steepest increase of the cost function using the *Chain Rule* presented in the previous chapter. Inverting the direction of that vector the Gradient Descent is found. The calculated vector is then used to update all weights and biases.

4. CNN applied to colored images and board games

4.1. 3D and multi-dimensional input. Up until now, the input to the Convolutional Neural Network has been a matrix of gray-scaled values without depth. In

reality, images have 3 channels, one for color Red, one for color Green and one for color Blue. The values at each channel can vary from 0 to 255, representing the intensity of that color for a particular pixel. This means that the overall dimensions of the input image are $X \times Y \times 3$, where X and Y are the length and width, and 3 is the depth. In such cases, there are two workarounds possible.

2D convolutions with 3D input: The first one will use 3 dimensional kernels as well. This means that now the kernel also has depth, which will be the same depth as the input depth. In the case of images, this means that the kernel of the first hidden layer will be of size $X \times Y \times 3$. This will let the first hidden layer produce the same output as if it was a 2 dimensional input. In such a workaround, then, the only difference would be in the first hidden layer.

3D convolutions with 3D input: In this second case, the kernel used can be also 3 dimensional, but its depth will be less than that of the input image. In this case, the kernel will slide not only across the length and width of the image, but also across its depth, generating a 3 dimensional feature map. In the particular case of images, this means that the kernels in the first hidden layer can not only detect object patterns, but also color patterns throughout all possibilities that the RGB color schema allows.

The same workarounds can be used in multi-dimensional convolutional neural networks. There may be problems in which the input to a convolutional neural network is a 4-D data representation. Adding one dimension to the kernel will solve the dimensionality problem.

4.2. Convolutional Neural Networks in Board Games. Once the inner workings of a Convolutional Neural Network and why it is able to detect patterns in images are understood, the transition to board games is very simple. When training a CNN to play a board game (Go, Chess, Checkers, Backgammon, etc) the input to the network will be the board state represented in such a way that a computer can understand it. The network will be trained with thousands of different board states while learning to depict the important patterns to make a good move. In this case, instead of learning patterns of colors and edges the network is learning patterns of chess pieces and movements.

Part 2

Practical Framework

Chapter 5
Practical project introduction and motivation

In this second part of the thesis, special focus will be given to the practical project executed along with the study of the concepts explained in the theoretical part. As explained in the introduction of this thesis, the practical project consisted in devising, developing and testing different neural network structures. Once this neural networks were programmed, they were trained to predict chess moves.

The idea of creating an artificial intelligence based on neural networks came after watching AlphaGo (the machine created by DeepMind) defeating the best Go player in the last decades. The game of Go has been the holy grail of artificial intelligence and neural networks for years. Although aiming to develop a machine like AlhpaGo is a very ambitious goal, trying to design a computer program that could potentially beat Chess players (giving it enough training time and power) was a very good target for various reasons. In the first place, it would require to read and understand not only the basics but also the inner workings of artificial neural networks and convolutional neural networks, an area of study still growing, highly useful and continuously being adopted by big companies and researchers. Secondly, a huge amount of programming is required to develop the framework that ease the communication between neural networks and chess databases, which in turn require the use of up to date technical programming libraries.

1. Project goals

The initial idea for this project was simply to develop a single convolutional neural network that could potentially learn to play chess and beat human players. As stated in this project introduction, this can be a very ambitious goal for various reasons:

- The time required to develop and program the neural network can take months. Different neural networks structures must be tested before selecting one that could potentially learn to play chess. Each structure needs to be properly trained and tested.

- There are other algorithms in play that need to be devised for the machine (not only the ones that build the neural networks). For example, an evaluation function that gives a very well approximation of which player is currently in a better position is needed before and after feeding the neural network. This evaluation function is used in a tree-search algorithm, like Monte-Carlo simulations, that allows the machine to see various moves ahead of time and gives the machine the ability to select the branch that will give the higher probability of winning.
- The computational power needed to train such neural networks is huge compared with the available computer power resources in the lab. The version of AlphaGo that beat the european Go championship Fan Hui used 147 GPUs, and the version that beat world Go championship Lee Sedol used 48 TPUs (*Tensor Processing Unit*) developed by Google [**15**] and specially designed for machine learning algorithms.

After realizing that the requirements needed to develop such project were that big, the goal was slightly modified to adapt it to the available infrastructures and personal interest. It was changed so that the project goal aimed to something that could allow for the learning, rather than the sophistication. This is how the main goals for this project evolved. The final goals are listed below:

- Design, develop and program a framework to ease the communication between chess binary representations and neural networks.
- Devise different Neural Network structures that could potentially learn to play chess, although being able to beat a human player wasn't a requirement. Instead, studying how and why the neural networks learned to play chess was the main objective.
- Design and apply statistics to better discern which Neural Network structure works better and why.

In other words, the main objective for the Neural Networks was not to play chess correctly, but simply to learn how it is played. There were no initial conditions given to the neural networks, such as how each figure moves or even that the figures should be moved. Neither was given the number 1 rule of chess: kill the opponent's King.

This means that the neural networks had to learn all this from the professional games shown to them. Needless to say, given that no preconditions nor rules of how the chess is played were given to the networks, the first move predictions had no sense at all. This will be further discussed in the last part of this thesis, *Results and Conclusions*.

The process of learning how the game of chess is played can be divided in the following major steps:

- Understand that there are different figures in the board, and that only figures can be moved (a board square without a figure is not valid as a starting point for a move).
- Once the machine learns that the figures should be moved, it should understand that each figure moves in a particular way.

- Once the machine learns how each figure moves, different strategies should be learned to kill the opponent's King.

This are the learning steps for the neural networks initially expected. As it will be discussed in the *Results and Conclusions* chapter, the neural networks trained could reach the second stage.

2. Project infrastructure

The practical project has been developed using a personal laptop, although the neural network models have run in B2SLab's Graphical Processing Units (GPUs) to increase efficiency during the training process. In particular, three GPUs were used: 2 Gefore GTX 1070 and a Titan Xp. But, as these GPUs were from the B2SLab's department, they were also used by other researchers and PhD students, which made them unavailable in some cases. The lack of GPUs and their disponibility has been a huge downside in the execution of this project. This is because the networks created need a lot of computer power for a long time during the training process.

As previously mentioned, the programming languages used are mostly Python and R. Python has been used to develop the framework that allows the communication between the Neural Networks and the Chess game logic. In other words, the Chess games used to train and test the neural networks are fed to the Neural Network through the Chess game logic implemented using Python. The Neural Networks in turn have also been developed using Python, but not all the structure and mathematical algorithms have been programmed. Instead, third party APIs (or Python libraries) have been used. In more detail, the two main libraries used for Deep Learning are Keras [**3**] and Tensorflow [**4**]. Tensorflow is an open source software library originally developed by Google for high performance numerical computation. Quoting from Tensorflow website:

Its flexible architecture allows easy development of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researches and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

Although one of the main goals of tensorflow is to allow for Deel Learning development, its main focus is numerical computation (which is a requirement for deep learning algorithms).

Keras, in the other hand, is a high-level neural networks API, specifically build for Neural Networks and Deep Learning. It is a wrapper around other more sophisticated libraries focused in computation efficiency rather than easy usage. These more sophisticated libraries are Theano, CNTK and **Tensorflow**. The guiding principles of Keras as are presented in their website are:

- **User friendliness.** Keras is an API designed for human beings, not machines. It puts user experience front and center. Keras follows best practices for reducing cognitive load: it offers consistent and simple APIs, it minimizes the number of user actions required for common use case, and it provides clear and actionable feedback upon user error.
- **Modularity.** A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as few restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.
- **Easy extensibility.** New modules are simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.
- **Work with Python.** No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

In other words, Keras hides the complexity of Tensorflow and specializes in user interaction in favor of usability. This is way Keras has been used to develop the neural networks trained in this project. Also, Tensorflow was used as its backend because of its performance and extensive usage in other deep learning projects.

Another reason why Tensorflow was selected as the backend for Keras is that Tensorboard was used during the training process to easily visualize how the model trained. Tensorboard was developed by the Tensorflow team. Thus, it can only be used in conjunction with Tensorflow. Tensorboard is a suite of visualization tools to make it easier to understand, debug, and optimize TensorFlow programs. Tensorboard runs in the background, reading the models continuously being saved in a computer disk directory during the training process.

Finally, once the models were trained, R was used to test and analyze the performance of such. From this tests and analysis, further conclusions were taken. To automatize this process, Bash was used, which helped when dealing with movement of files between directories.

3. Project methodology

Once the infrastructure was assessed, the project guidelines were established. The first thing that had to be done was to get the data used during the process of training and testing the models. The data had to be split into three different subsets: the training, validation and testing subsets. The validation subset is used during the training process to supervise the models and check for possible overfits. The test subset was used during the final analysis of the model.

After the data had been downloaded, the programming structure had to be devised and programmed. A basic class was used to simulate the logic behind a chess board.

Another class was used to read and parse the chess data files, which were translated to binary game representations using the chess logic class.

Once the Logic Class and the Parser Class had been programmed, various neural networks were developed and trained during the course of 2 to 3 months. This various neural networks included some Artificial Neural Networks and various Convolutional Neural Networks. The Artificial Neural Networks were used for testing purposes, which helped in several ways:

- They helped solving some major problems and minor bugs present in the Logic and Parser classes.
- They allowed for the understanding of how Keras and Tensorboard work and how neural networks are implemented with this libraries.
- They represented a foothold for further and more complex neural networks.

After the initial experiments were made, the final configuration of neural network structures were devised to be compared between them. Because this project focuses mainly on deep learning, the 5 neural networks compared where convolutional neural networks.

Finally, the R script responsible for evaluating each model was developed.

Chapter 6
Data collection and representation

Chess is a very complex board game. Not only it has 64 different positions in the board but there are also a total of 32 initial pieces. Each of these pieces belong to a certain figure type. The 6 figure types are:

- Pawn (P): 8 Pawn pieces start in the front row of a player.
- Rook (R): 2 Rook pieces start in the corners of the players board side (back row).
- Knight (N): 2 Knight pieces start next to the Rook pieces.
- Bishop (B): 2 Bishop pieces start next to the Knight pieces.
- Queen (Q): 1 Queen is positioned in the center square of the back row that has the same color as the players color (black or white).
- King (K): 1 King is positioned in the remaining square.

Because of this complexity, coming up with a way to represent a whole Chess game (from start to end) can be difficult. That's why a universal plain text computer-processible format for recording chess games was developed in 1993 by Steven J. Edwards [**16**]. The format is called Portable Game Notation (PGN) and it has two main conveniences. In the first place, it allows to represent a whole Chess game from start to end in a few lines of text. Moreover, this compact way of saving chess games is perfect for computer programs that are trained to play chess.

1. Portable Game Notation (PGN)

A chess game represented in a PGN file looks like this:

```
[Event "World Cup"]
[Site "?"]
[Date "2013.08.11"]
[Round "?"]
[White "Gelfand, Boris"]
[Black "Rahman, Ziaur"]
[ECO "A40"]
[WhiteElo "2764"]
[BlackElo "2470"]
[Result "1-0"]

1. d4 e6 2. c4 d5 3. Nf3 Nf6 4. Nc3 Bb4 5. Bg5 h6 6. Bxf6 Qxf6 7. e3 O-O 8. Rc1 dxc4
9. Bxc4 c5 10. O-O cxd4 11. Ne4 Qe7 12. exd4 Nc6 13. Qe2 Bd7 14. a3 Bd6 15. Rfd1
Rad8 16. Qe3 Rfe8 17. b4 a6 18. Be2 Nb8 19. Ne5 Ba4 20. Rd2 Bxe5 21. dxe5 Rxd2 22.
Qxd2 Rd8 23. Qb2 Bc6 24. Nd6 Bd5 25. f4 Nc6 26. a4 Qc7 27. b5 Qb6+ 28. Kf1 axb5 29.
axb5 Ne7 30. Qc3 Rf8 31. Qc5 Qa5 32. Rd1 Qa2 33. g3 f6 34. exf6 Rxf6 35. Qc3 Qa7 36.
Ne8 Rf7 37. Bh5 g6 38. Nf6+ Rxf6 39. Qxf6 gxh5 40. Qxe7 Bc4+ 41. Kg2 Bd5+ 42. Kh3
Qa2 43. Rxd5 Qxd5 44. Kh4 Qxb5 45. Qxe6+ Kg7 46. f5 Qc6 47. Qe7+ Kg8 48. Kxh5 b5 49.
g4 b4 50. Qxb4 Qc7 51. Qb3+ Kh8 52. Qe6 Qf7+ 53. Qg6 Qc7 54. Qxh6+ Kg8 55. Qe6+ Kh8
56. Qe8+ Kh7 57. h4 Qb7 58. Qg6+ Kh8 59. Qh6+ Kg8 60. Qe6+ Kh8 61. g5 Qf7+ 62. g6
Qf8 63. g7+

1-0
```

Fig. 1

There are three parts in a single pgn game. The first one is called *Tag pair section*. This is the first section seen in the image, where each line is encapsulated with square brackets. This section gives general information of the chess game represented. Information such as when and where this game was played, who where the players and their rank. For this thesis, this first part of every pgn game was of no use and was programmatically discarded.

The second part is the one used in this project. It contains all moves that took place in the game. The move number is indicated in front of every white move. In other words, move 1 of the game contains the first move for White and the first move for Black. The game represented in the image above, as an example, contains 63 moves.

To represent each move, a representation standard using ascii latin characters is used, called Standard Algebraic Notation (SAN). An alternative of SAN is FAN (Figurine Algebraic Notation), which uses miniature figure icons rather than the letter nomenclature to represent each piece.

In SAN notation, the chess board squares are represented with a letter-value pair. The letter represents the column (starting in the bottom left corner of a chess board) and the number indicates the row.

A basic SAN move is given by listing the moving piece letter capitalized (omitted for pawns) followed by the destination square. If an enemy figure is captured, it is represented by the lowercase *"x"* before the destination square. If a pawn is the killer figure, its column is inserted at the front. Following are a couple example:

- **d4**: Because it has no capital letter, this movement is a Pawn movement. The destination square is *d4* which means that the initial position must be *d2* or *d3*, considering the legal moves for Pawns.
- **Rxf6**: The figure moving is a Rook. Because there is an *"x"* prior to the destination square, this movement involves a kill.
- **dxc4**: Again, the figure moving is a Pawn because there is no capital letter representing the figure. Because there's an *"x"*, this move involved a kill at location *c4*.

The king's castle movements are indicated with multiple *"O"*, being *"O-O"* the kingside castling and *"O-O-O"* the queenside castling.

Pawn's promotion are indicated with an equal sign *"="* immediately following the destination square and followed by the promoted piece letter (indicating one of knight, bishop, rook, or queen). Example: **e7=Q**, which indicates the promotion of a Pawn at location *e7* to the queen figure.

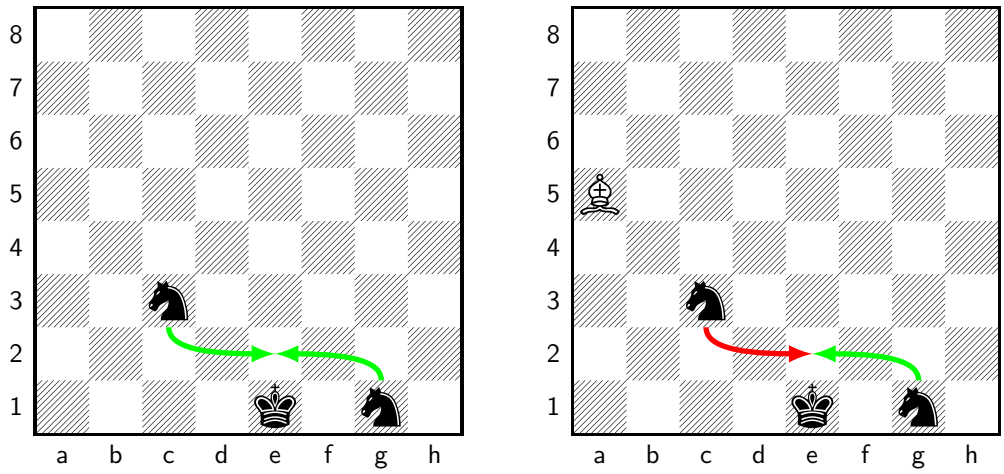
If the move is a checking move, the plus sign "+" is appended as a suffix to the basic SAN move notation; if the move is a checkmating move, the octothorpe sign "#" is appended instead.

1.1. SAN Disambiguations. There may be cases with ambiguity. For example, when two pieces of the same type can go to the destination square represented in the SAN notation. To solve this ambiguities, the following three steps are taken in order, starting with the first one:

(1) First, if the moving pieces can be distinguished by their initial position's column, the initial position's column of the moving piece is inserted immediately after the moving piece letter. Example: **Rfe8** indicates a Rook move from column *f* to square *e8*.
(2) Second (when the first step fails), if the moving pieces can be distinguished by their initial position's row, the initial position's row of the moving piece is inserted immediately after the moving piece letter. Example: **B3d4** indicates a Bishop move from row 3 to square *d4*.
(3) Third (when both the first and the second steps fail), the two character square coordinate of the originating square of the moving piece is inserted immediately after the moving piece letter. Example: **Nb1c3**.

This disambiguation steps are only used when two figures of the same type can go to the destination square using legal moves. An example of a move that might seem ambiguous but it is not is show in the next diagrams.

There's an empty square at *e2* that both Black Knights can occupy. In the left diagram both moves are legal, thus the SAN notation will have to be *Nce2* or *Nge2* to differentiate between Knights. Nonetheless, in the right diagram, the Knight at position *c3* can't move to *e2* because it would leave the Black King in checkmate. Thus, there's only one possible Knight that can go to *e2* in this situation, and the SAN notation would be *Ne2*.



The SAN move length can vary from just 2 characters when moving an initial Pawn to several characters, with a maximum of 7. A maximum length SAN move can be, for example, **Qa6xb7#** which means: the Queen at position *a6* kills the enemy figure at position *b7* and checkmates its opponent. Another example could be **fxg1=Q+** which means: the Pawn at column *f* kills the enemy figure at location *g1*, which implies a promotion to Queen, all ending in a check for the opponent.

Understanding the PGN notation was crucial in the practical project because, although there are existent Python libraries that can automatically read PGN files, a special parsing class had to be developed. This class helped translating chess moves from PGN notation to Convolutional Neural Network's input representation.

2. Chess game state representation

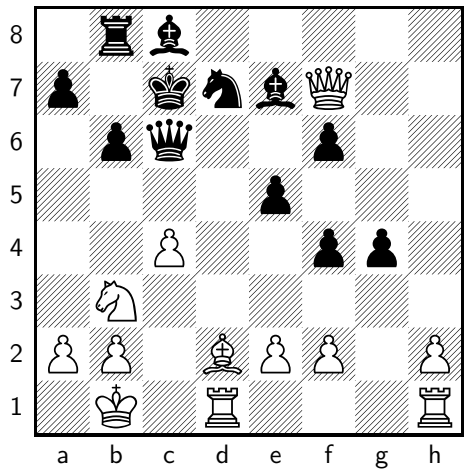
As previously mentioned, computers don't understand PGN notation directly. PGN notation was made to ease the parsing of chess games for developers, but a neural network can't have an input of an *"N"* (for Knight). What would it mean?

Neural networks need real number inputs, or at least binary inputs. For this project, a special representation involving binary matrices in several dimensions was used. The idea behind this kind of representation was to feed the network with as much information as possible in multiple dimensions.

Two different representations have been employed. One for the neural networks used during the first tests and another one used with the final convolutional neural networks.

The dimensions of the first representation used for the test neural networks were $2 \times 7 \times 8 \times 8$. This $2 \times 7 \times 8 \times 8$ multidimensional matrix represented one single state of a chess game (one position of the board during a game). When the input matrix for a neural network surpasses the 3 dimensions we are used to, it's better

to represent it as a tree, and not a matrix. The following diagram shows a board position in the middle of a game. This board position is used to show the binary representation of such position.



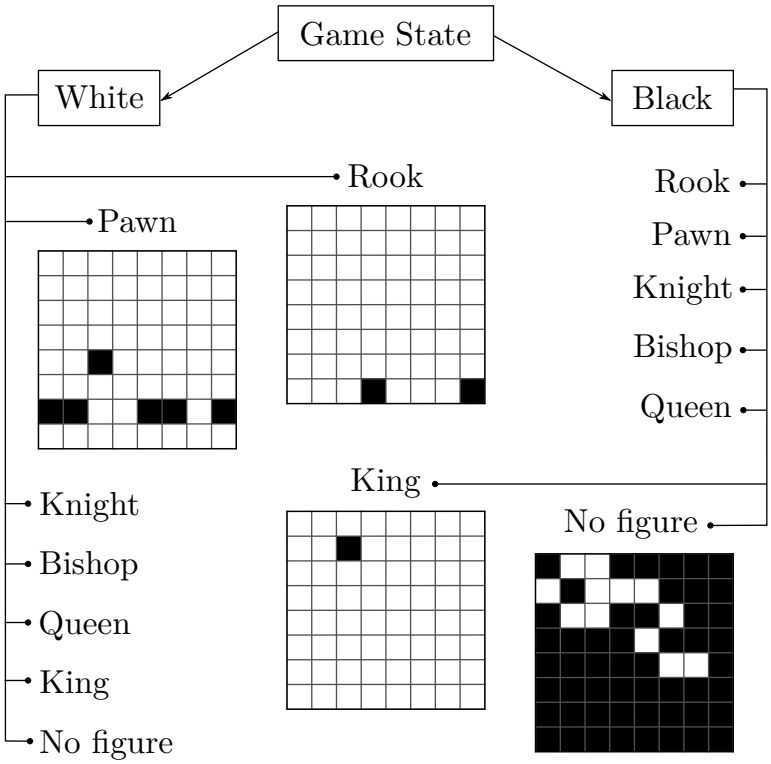


Fig. 2

In the shown diagram, black squares represent a 1 in binary, while white squares represent a 0. Using black and white squares is better to visualize the tree. As previously explained, the input matrix has 4 dimensions ($2 \times 7 \times 8 \times 8$). The first dimension is the player (*Black* or *White*). The second dimension represents the type of figure (each player has 7 possible figures). Notice that besides the 6 pieces normally used in chess, a 7th type was added (*No figure*) to add extra information. The third and fourth dimensions represented the board matrix with *1*'s (black squares in the diagram) where the figure is present and *0*'s (white squares in the diagram) otherwise.

Because Artificial Neural Networks don't understand multidimensional inputs (they have a flattened input layer), this representation had to be flattened before being fed to Artificial Neural Networks. This means that the multidimensional matrix with $2 \times 7 \times 8 \times 8$ dimensions becomes a single vector of size 896.

For the convolutional neural networks used in the final stages of this project, a different representation was used. This representation has been previously used in other studies [**17**]. This time, the input data dimensions were $8 \times 8 \times 6$. This data representation is better to comprehend because it resembles the one used with images. The first two dimensions represent the board matrix, and the last dimension is a one-hot vector of length 6 (representing the figure type). To distinguish between black figures and white figures, the black ones were negated. Thus, instead of having a 1 representing the black figure, -1 was used. The following diagram shows a visual representation of this method for the board position previously shown.

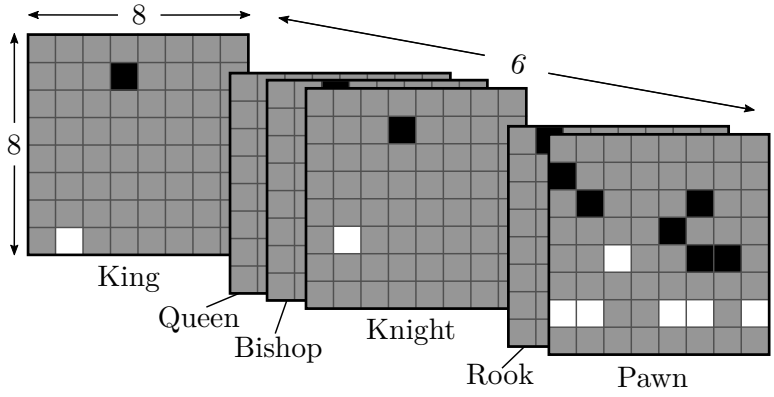


FIG. 3. This figure shows the data representation used by the convolutional neural networks. It consists of an $8 \times 8 \times 6$ matrix. The first two dimensions represent the chess board. The last dimension corresponds to the number of figures. The black squares are -1 representing a black figure in that position. White squares are 1's in the matrix, representing white figures.

Both types of data representation were used in this project. The first one with the initial neural networks and the second with the final convolutional neural networks. In the following chapter, the structure of such networks will be discussed.

3. Databases used

To train and test the models created, multiple databases have been used. These databases come from online sources that contain thousands of games played by professionals during chess tournaments. The games are annotated using PGN notation and are available for free.

In particular, two sources have been used. One is a dataset of games called *gmall-both.pgn* and downloaded from *chess-db.com* [**18**].

Another source of games used is from a website called *KingBase* [**19**], which offers a free chess database uploaded monthly.

In total, over 2,000,000 chess games were used. This games were further split into three groups, the training dataset, the validation dataset and the test dataset.

It's important to notice that the neural networks developed are trained to predict moves. This means that with more than 2,000,000 chess games and an average of 40 moves per game [**20**], there were a total of 80 million moves to train the networks with.

Chapter 7
Neural Network's output data fromat

In this chapter, an overview of the output representation for the neural networks created will be given.

As stated initially in this practical framework, the main goal for this project was to develop neural networks that were capable to learn how to play chess (not to actually play it right). This implies that the neural networks had to be able to predict moves. In other words, the output layer of the neural networks used had to represent, in some way, a move in the chess board.

When developing a machine learning algorithm to play chess, all possible actions taken by a player in its turn must be considered. When it is a player's turn, this can perform the following actions:

- **Resign**: If the board position is very unfavorable for the player, this can abandon the game.
- **Normal move**: This action includes all possible moves that a player can perform in its turn. Castling is treated as a normal chess move.
- **Pawn promotion**: When a *Normal move* implies moving a pawn to the last row of the board, a promotion of this pawn to one of the following figures is permitted: Queen (Q), Knight (N), Rook (R), Bishop (B). This means that the player must specify which figure to promote the pawn to.

In other words, for a neural network to be able to play chess like a human, it must be capable of performing all this actions. In other studies, multiple neural networks were used to create a machine capable of all this actions. For example, a single neural network can be created to evaluate the board position and output a single value telling which player is more likely going to win. This single neural network can then be used as an indicator that tells whether the machine should resign or not.

Another neural network could be trained to perform pawn promotions. Given a board position with a pawn promotion in place, this neural network would tell which figure to promote the pawn to. This neural network would probably be overfitted

if not trained correctly because in 98% of promotions performed by professionals the promoted figure is the *Queen* [**21**].

Finally, one or multiple neural networks could be trained to perform normal moves.

For this project, the initial idea was to create a single neural network that outputted all possible actions that a player can perform. This is not a good approach, because all actions would be evaluated by the same network. The network would be predicting a knight move and deciding to resign at the same time. This doesn't make sense. When a human player decides to resign in a game, it does so evaluating only the board position and not the possible moves. In other words, when a human is playing, it decides whether it should resign or not independently of which moves his/her knight can play.

Once this initial idea was considered a bad approach, and given that the main goal of this project was to see how a neural network learns the rules of chess and not to win human players, only *Normal moves* were considered. A single neural network was created to predict a chess move given a board position, nothing else. This approach fits with the project premise: the neural network has to learn how chess is played form scratch.

1. Output representation

The output representation of the neural networks used consists of 32 neurons divided in 4 classes. This means that the neural network is dealing with multi-label multi-class classification (as seen in the theoretical framework).

The first 2 classes represent the initial position of the predicted move. The last 2 classes represent the final position of the predicted move. All classes have 8 labels (representing a row/column).

Once a move is predicted, the first 8 output values are read. This first 8 values will tell which column is the initial column. Then, the values from the neurons number 9 to 16 are also read, which will tell the initial row. Once the initial position is known, the final position is also read from the last 16 values in the same manner.

The following matrix represents an output of the neural network. Each row represents one class. The first matrix row represents the initial position's column, the second matrix row represents the initial position's row, the third row represents the final position's column and the fourth row represents the final position's row.

$$\begin{bmatrix} 0.23 & 0.89 & 0.56 & 0.33 & 0.07 & 0.15 & 0.10 & 0.40 \\ 0.13 & 0.10 & 0.09 & 0.30 & 0.38 & 0.98 & 0.72 & 0.31 \\ 0.55 & 0.70 & 0.65 & 0.88 & 0.12 & 0.77 & 0.61 & 0.66 \\ 0.90 & 0.11 & 0.01 & 0.12 & 0.27 & 0.49 & 0.38 & 0.22 \end{bmatrix} \tag{38}$$

This means that, given this output, the predicted move is $(2, 6) \rightarrow (4, 1)$. Which means: move the figure present at position $(2, 6)$ to position $(4, 1)$.

With this output representation it becomes clear that the predicted moves don't take into consideration the type of figure moving (if moving a figure at all) and how it moves. In other words, the neural network will start performing random moves from random positions to random positions, and its goal is to slowly learn the rules of chess.

Chapter 8
Neural Networks used during the practical project

As previously exposed, during the first months different Artificial Neural Networks were tested to get an idea of how the framework developed and the libraries used worked. During this process the main goal was to polish the framework of Python classes used to communicate with the neural networks.

After trying different small configurations for the initial neural networks, a big Artificial Neural Network was used to see the first important results of this project. In the following section, this artificial neural network will be discussed.

1. First Artificial Neural Network

The first big neural network used for testing was a feed-forward MLP or Artificial Neural Network. The input data used for this neural network was the first one described in the previous chapter. That is the one with 4 dimensions and a total of 896 binary values. Because this was a MLP, the input layer didn't have multiple dimensions and the input data had to be flattened to a vector of size 896. This means that the input layer for this network had 896 neurons.

A summary of this network from the python library *Keras* can be seen in the next page:

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 896)	803712
dense_2 (Dense)	(None, 853)	765141
dropout_1 (Dropout)	(None, 853)	0
dense_3 (Dense)	(None, 810)	691740
dense_4 (Dense)	(None, 766)	621226
dropout_2 (Dropout)	(None, 766)	0
dense_5 (Dense)	(None, 723)	554541
dense_6 (Dense)	(None, 680)	492320
dropout_3 (Dropout)	(None, 680)	0
dense_7 (Dense)	(None, 637)	433797
dense_8 (Dense)	(None, 594)	378972
dropout_4 (Dropout)	(None, 594)	0
dense_9 (Dense)	(None, 550)	327250
dense_10 (Dense)	(None, 507)	279357
dropout_5 (Dropout)	(None, 507)	0
dense_11 (Dense)	(None, 464)	235712
dense_12 (Dense)	(None, 421)	195765
dropout_6 (Dropout)	(None, 421)	0
dense_13 (Dense)	(None, 378)	159516
dense_14 (Dense)	(None, 334)	126586
dropout_7 (Dropout)	(None, 334)	0
dense_15 (Dense)	(None, 291)	97485
dense_16 (Dense)	(None, 248)	72416
dropout_8 (Dropout)	(None, 248)	0
dense_17 (Dense)	(None, 205)	51045
dense_18 (Dense)	(None, 162)	33372
dropout_9 (Dropout)	(None, 162)	0
dense_19 (Dense)	(None, 118)	19234
dense_20 (Dense)	(None, 75)	8925
dropout_10 (Dropout)	(None, 75)	0
dense_21 (Dense)	(None, 32)	2432

Total params: 6,350,544
Trainable params: 6,350,544
Non-trainable params: 0

FIG. 1

The *summary* function from *Keras* is a great tool that helps visualizing the structure of the neural networks built.

As can be seen from the summary above, the first neural network consisted of a total of 31 layers. These layers consist of a combination of Dropout and Dense layers. In *Keras*, Dense layers are those that maintain a fully connected structure between the layers above and below it. In other words, all neurons in this layer are connected to all neurons in the previous and next layers. Dropout layers were discussed in the theoretical framework. Dropout neurons randomly deactivate during the training process to reduce overfitting (it's a regularization technique).

The activation function used for all neurons in this network is the $ReLU$ function.

The second column shows the number of neurons present at each layer. Notice how the trend in this particular neural network is for the number of neurons in consecutive layers to go down. This means that the final structure of such network can be thought as an inverted triangle. This type of structure allows for the layers to just keep with the information really important from previous layers. Another type of structure is the hourglass shape presented in the theoretical framework, which allows for extraction of information not directly present in previous layers.

The total amount of trainable parameters is $6,350,544$, which means that the parametric space has $6,350,544$ dimensions.

This neural network was left training for almost 4 days with the training dataset and the validation dataset presented in the previous section.

2. Convolutional Neural Networks

For this project, 5 convolutional neural networks were trained. One of the goals for this project was to see how different convolutional neural structures perform. In order to observe this, small changes were applied to a base convolutional neural network structure. This small changes will then translate to small changes in performance and accuracy.

The basic convolutional neural network had the following hyper-parameters[1]:

- **Number of convolutional layers**: 8
- **Activation function for convolutional layers**: $ReLU$
- **Number of kernels per convolutional layer (8)**: $128-128-256-256-512-512-1024-1024$
- **Kernel sizes (8)**: $8-8-6-6-4-4-2-2$
- **Number of pooling layers**: 3
- **Size of pooling layers (3)**: $3-3-2$
- **Number of fully connected layers**: 4
- **Activation function for fully connected layers**: $ReLU$
- **Number of neuron per fully connected layer (4)**: $2048-1024-512-32$

[1]The word hyper-parameters is used to distinguish between the parameters that are trainable and the ones that are not. Hyper-parameters are the second ones.

Once this first model was trained, 4 new models were created with slight changes from this initial structure. The main ambition for this methodology was to see how the 2 main components of a convolutional neural network affect the resulting outcome. These two main parts are: the feature recognition part and the fully-connected neural network part. Thus, the slight changes applied to these 4 new models modify the 2 components of the base CNN.

2.1. Fix convolutional layers, less dense layers. This convolutional neural network had the same amount of convolutional layers (the same amount of feature recognition layers) as the previous model, but with less fully-connected layers. The following hyper-parameters were used to train this model:

- **Number of convolutional layers**: 8
- **Activation function for convolutional layers**: $ReLU$
- **Number of kernels per convolutional layer (8)**: $128-128-256-256-512-512-1024-1024$
- **Kernel sizes (8)**: $8-8-6-6-4-4-2-2$
- **Number of pooling layers**: 3
- **Size of pooling layers (3)**: $3-3-2$
- **Number of fully connected layers**: 2
- **Activation function for fully connected layers**: $ReLU$
- **Number of neuron per fully connected layer (2)**: $512-32$

The only changes when compared to the base structure are the number of layers for the fully-connected network. It uses only 2 layers instead of 4.

2.2. Fix convolutional layers, more dense layers. This convolutional neural network had, again, the same amount of convolutional layers as the base model. Instead, this new model had more fully-connected layers. The following hyper-parameters were used:

- **Number of convolutional layers**: 8
- **Activation function for convolutional layers**: $ReLU$
- **Number of kernels per convolutional layer (8)**: $128-128-256-256-512-512-1024-1024$
- **Kernel sizes (8)**: $8-8-6-6-4-4-2-2$
- **Number of pooling layers**: 3
- **Size of pooling layers (3)**: $3-3-2$
- **Number of fully connected layers**: 8
- **Activation function for fully connected layers**: $ReLU$
- **Number of neuron per fully connected layer (8)**: $8192-4096-2048-1024-512-256-128-32$

This new model had 8 fully-connected layers instead of just 4.

2.3. Less convolutional layers, fix dense layers. This convolutional neural network had the same amount of fully-connected layers as the base model. Instead, it had less convolutional layers. The following hyper-parameters were used:

- **Number of convolutional layers**: 4
- **Activation function for convolutional layers**: $ReLU$
- **Number of kernels per convolutional layer (4)**: $128 - 256 - 512 - 1024$
- **Kernel sizes (4)**: $8 - 6 - 4 - 2$
- **Number of pooling layers**: 3
- **Size of pooling layers (3)**: $3 - 3 - 2$
- **Number of fully connected layers**: 4
- **Activation function for fully connected layers**: $ReLU$
- **Number of neuron per fully connected layer (4)**: $2048 - 1024 - 512 - 32$

This new model had just 4 convolutional layers instead of 8.

2.4. More convolutional layers, fix dense layers. Finally, this convolutional neural network had the same amount of fully-connected layers as the base model, but it had more convolutional layers. These are the hyper-parameters used:

- **Number of convolutional layers**: 16
- **Activation function for convolutional layers**: $ReLU$
- **Number of kernels per convolutional layer (16)**: $128 - 128 - 256 - 256 - 256 - 256 - 512 - 512 - 512 - 512 - 1024 - 1024 - 1024 - 1024 - 2048 - 2048$
- **Kernel sizes (16)**: $8 - 8 - 8 - 8 - 6 - 6 - 6 - 6 - 4 - 4 - 4 - 4 - 2 - 2 - 2 - 2$
- **Number of pooling layers**: 3
- **Size of pooling layers (3)**: $3 - 3 - 2$
- **Number of fully connected layers**: 4
- **Activation function for fully connected layers**: $ReLU$
- **Number of neuron per fully connected layer (4)**: $2048 - 1024 - 512 - 32$

This final model used 16 convolutional layers instead of 8. The results for this model were unexpected. Because of the huge amount of time it needed for the training process, there were no satisfactory results. It will be discussed in the last part of this thesis.

Chapter 9
Practical project recap

Before diving into the results of this project, a brief recap of the work done will be given. The main goal of this project was to construct different convolutional neural networks to see how different structures cam impact the network output. These networks were trained to predict chess moves, but not to win chess games. In other words, given a board position, the networks will try to guess what figure should move to where.

First, a simple artificial neural network was used to understand how the overall working environment (the python libraries and tensorboard) performed. Right after, the convolutional neural network structures were devised. 5 different convolutional neural networks were trained, all based on an base structure. The small deviations applied to each convolutional neural network will be used to evaluate the effect of the pattern recognition part and the fully-connected part of each CNN. In the next part of this thesis the results will be discussed.

Part 3

Results and Conclusions

In this final part of the project, the results of the models trained will be presented, as well as a conclusion of how they performed.

As explained earlier, the models were left training for two weeks, although two of the 5 models could only be trained for 2 days. During this training process, every time the cost function of the model decreased, a copy of the model at that exact time was saved. Doing so left with different frames of the same model in its process of learning. This distinct frames were later used by R to evaluate each CNN. Also, Tensorboard was used during the training process to see how the accuracy and cost function of the model evolved.

Thus, this chapter is divided in two main sections. The first one focuses on how the accuracy and cost functions evolved for each model using the results obtained from Tensorboard. The second section will focus on how each model plays each figure, trying to look for correlations between figure move learning and model structure.

Chapter 10
Tensorboard results - Accuracy & Cost function

As explained earlier, Tensorboard is a program developed by the Tensorflow team. It is a tool that helps visualizing the training process of a model. It offers a more in-depth look of how the model is training. To do so, Tensorboard reads a special log file created by Tensorflow at every training step (after each mini-batch is trained). Among the tools that Tensorboard offers there is:

- The possibility to look at individual neuron weights and biases and how they evolve during the training process.
- The possibility to look at the model's accuracy and how it evolves.
- The possibility to look at the cost function of the model and how it changes at each training step
- It also offers the possibility to create special metrics that can be obtained during the training process from the log files.

All this in a very easy to use and visual environment. For this project, only the model's accuracy and cost function were used to evaluate and compare the different models. In the following section, the 5 convolutional neural networks will be discussed using Tensorboard graphs.

1. Fix convolutional layers, fix dense layers (base model)

As a recap, this is the base model from which 4 variants were created. This base model had 8 convolutional layers and 4 fully connected layers. This model could only be trained for 2 days due to a lack of available GPUs. Although it trained for only 2 days, it is enough time to see the potential of such a neural network and how it compares to the rest of the models. In the following image, the accuracy of the model is presented:

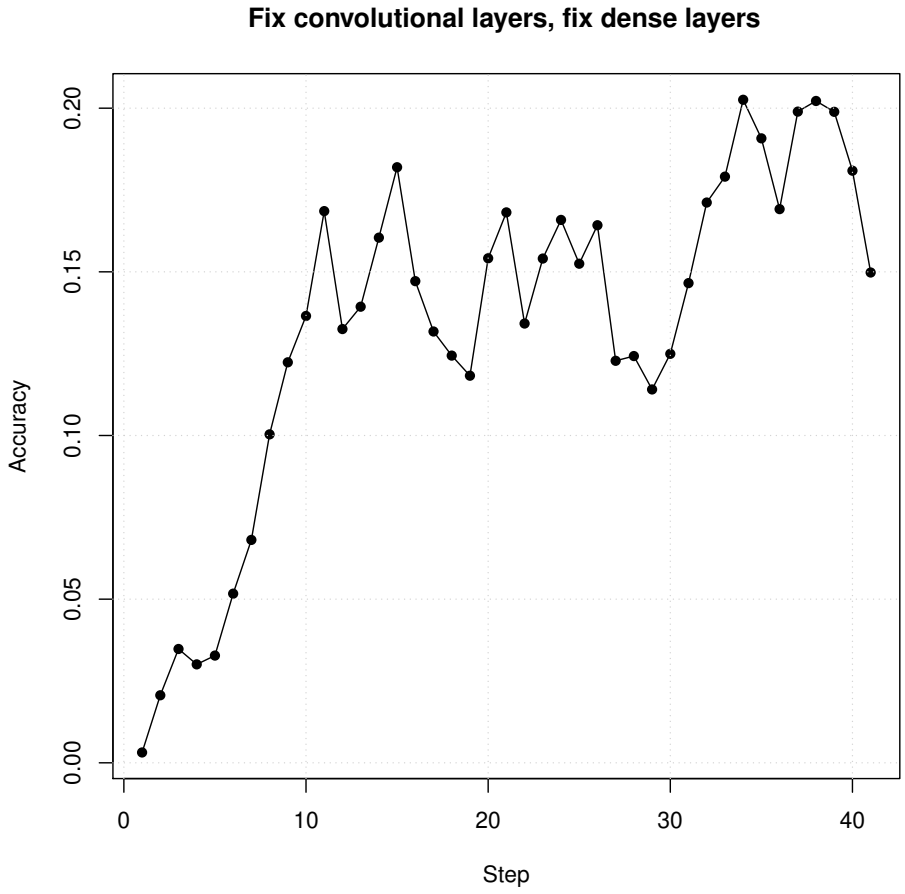


FIG. 1. This graph shows the accuracy over training time of the base convolutional neural network. The learning curve is very steep during the first training steps, and more gentle during the last steps.

The graph shows that eventually the learning of this neural network becomes more gentle, although it is convenient to remember that it only trained for 2 days. The highest accuracy is of 20%, which at first seems quite low.

Each step consisted of $100,000$ moves, given to the neural network in mini-batches of size 32. Once one step finished, the mean accuracy of all $100,000$ predictions was used to plot the graph shown above.

This means that this model, after training a total of roughly 4 million moves, managed to predict 2 out of every 10 moves correctly. It's noticeable to remember that this model, along with all other models trained, didn't know anything about chess before starting the training process. Having that in mind, 20% of accuracy doesn't seem so bad. Also important to consider the fact that chess games tend to

diverge a lot once the first 10 moves have be played. Finally, it's also important to add that this model had a total of $28,253,472$ parameters to be trained.

In the next section, when the R script is presented, a more in-depth analysis of the moves played by this model will be given.

The next graph shows the cost function of the model during the training process:

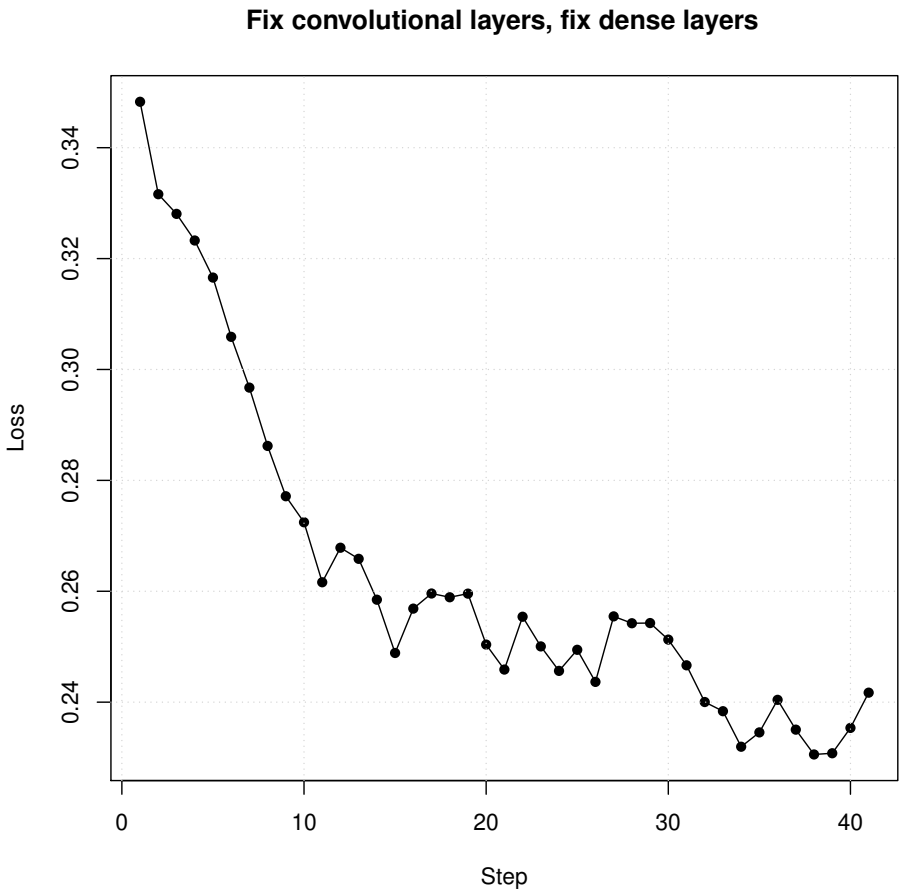


FIG. 2. This graph shows the cost function (or loss function) of the base model.

The loss graph correlates with the accuracy graph. There's a clear descent of the cost function during the first steps.

2. Fix convolutional layers, less dense layers

This model had the same structure as the base model except for the fully-connected layers which had 2 instead of 4. This model was left training for roughly 2 weeks. The following graph shows the accuracy of the model:

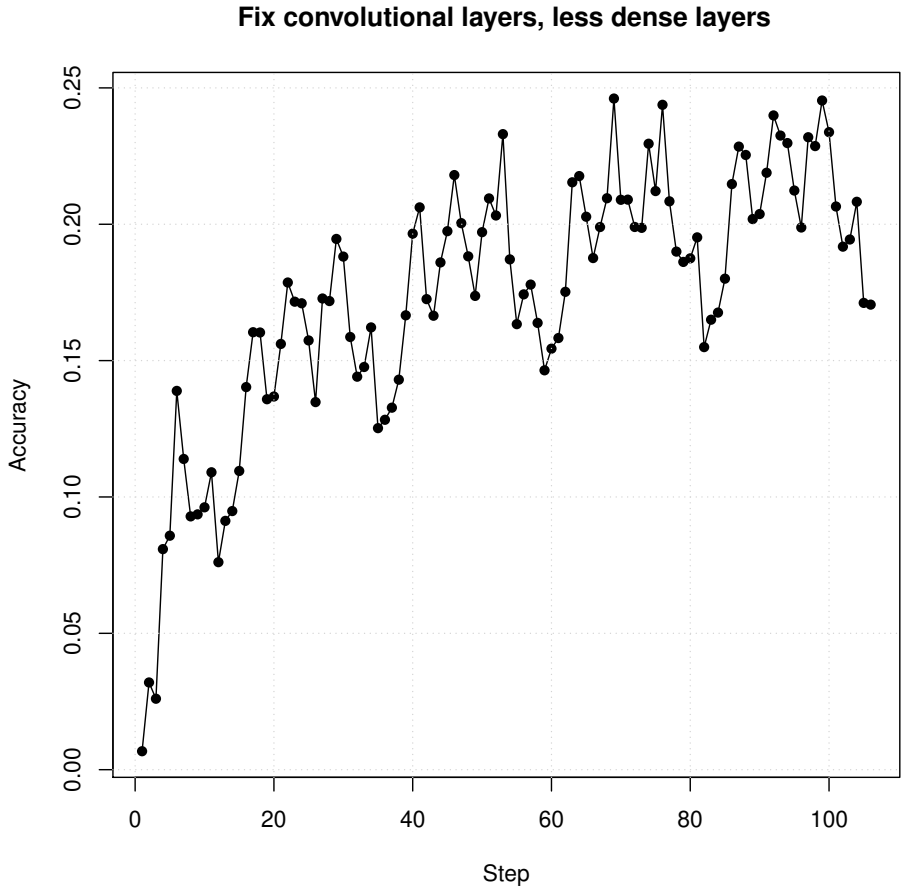


FIG. 3. This graph shows the accuracy of the "less dense" model.

Because this model had time to train for 2 weeks, it was able to train more than a 100 steps. Knowing that each step consists of $100,000$ moves, this model trained for a total of 10 million chess moves. After 2 weeks of training, the model managed to get to an accuracy of almost 25%. It doesn't seem much compared to the previous model, knowing that this had 1.5 times more training time. This type of logarithmic behaviour is quite common in machine learning and neural networks, where the accuracy increases rapidly during the first steps but in a more gentle manner as time increases. This directly relates to the learning algorithm used and presented in the theoretical framework.

This model had a total of $19,337,504$ parameters to be trained.

The following graph shows the evolution of the cost function for the "less dense" model:

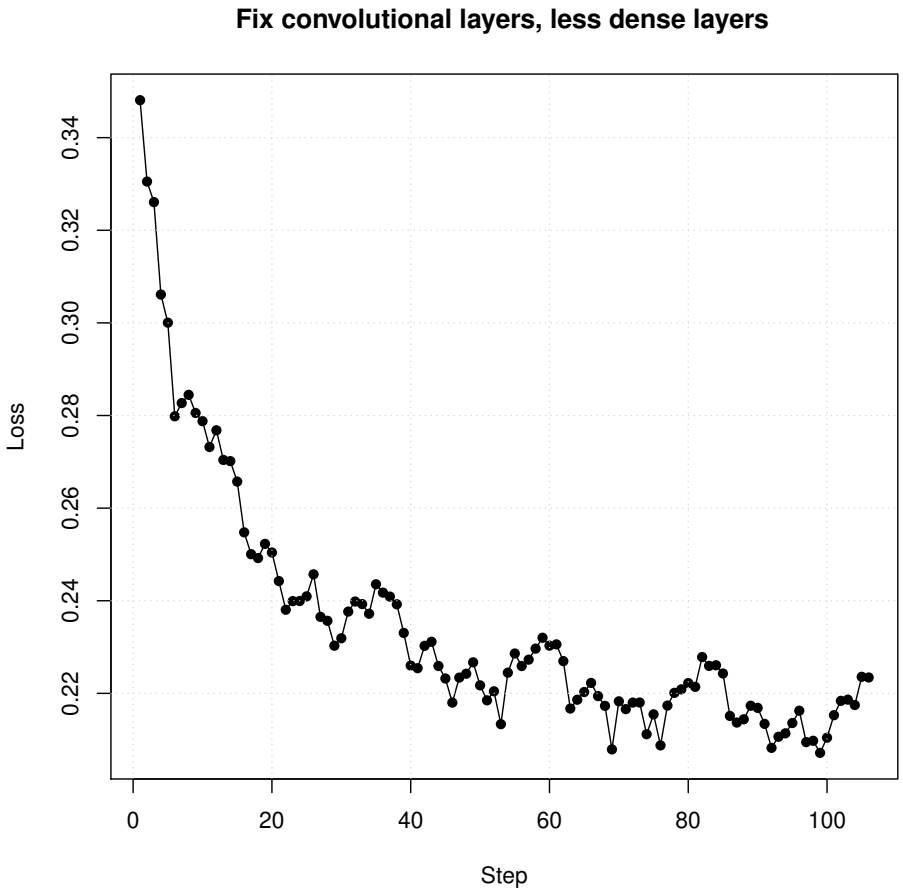


FIG. 4. This graph shows the cost function (or loss function) for the "less dense" model.

Again, a more in-depth look of the types of moves that this model predicts will be given in the R script section.

3. Fix convolutional layers, more dense layers

This model had the same structure as the base model but with 8 fully-connected layers instead of 4. This model also trained for roughly 2 weeks. The following graph shows the evolution of this model's accuracy during the training process:

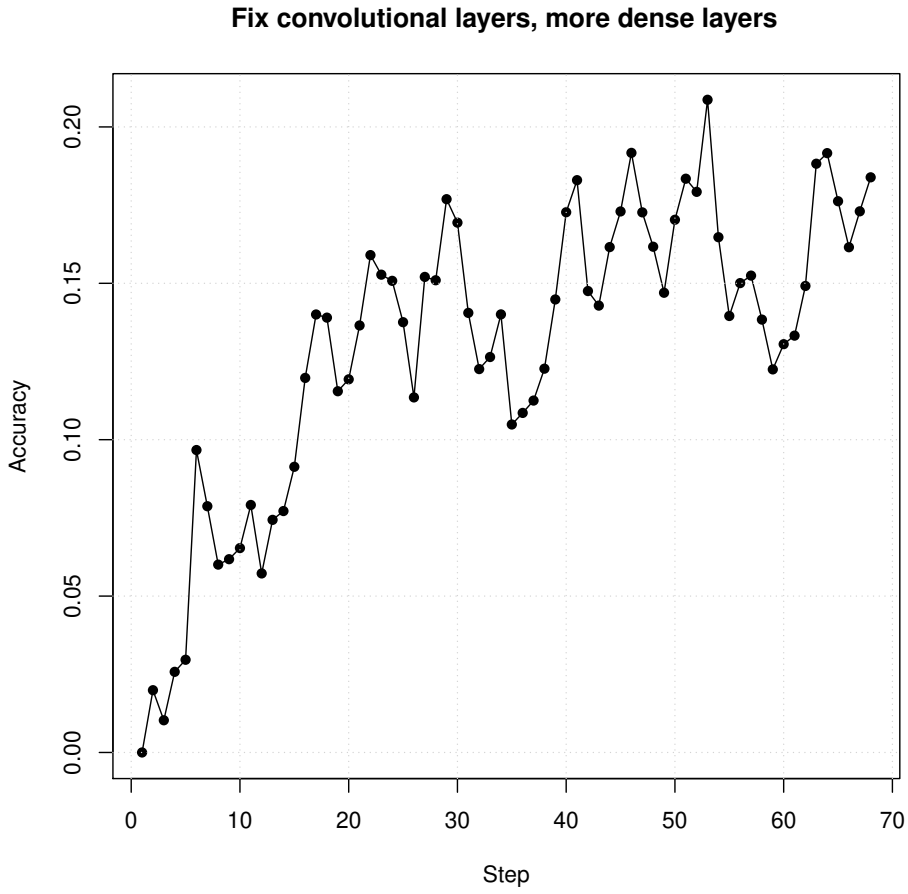


FIG. 5. This graph shows the accuracy of the "more dense" model.

Although this model also trained for 2 weeks, it only trained for almost 70 steps. This is because having more layers (more dense layers) implies having more weights and biases to be trained, which in the end correlates with more time needed for training. In particular, this model had $95,526,560$ trainable parameters, almost 5 times more than the previous layer.

During the training, it managed to get to a maximum of approximately 20% accuracy. Compared with the previous models it seems that this model fell behind.

The following graph shows the loss function of this "more dense" model:

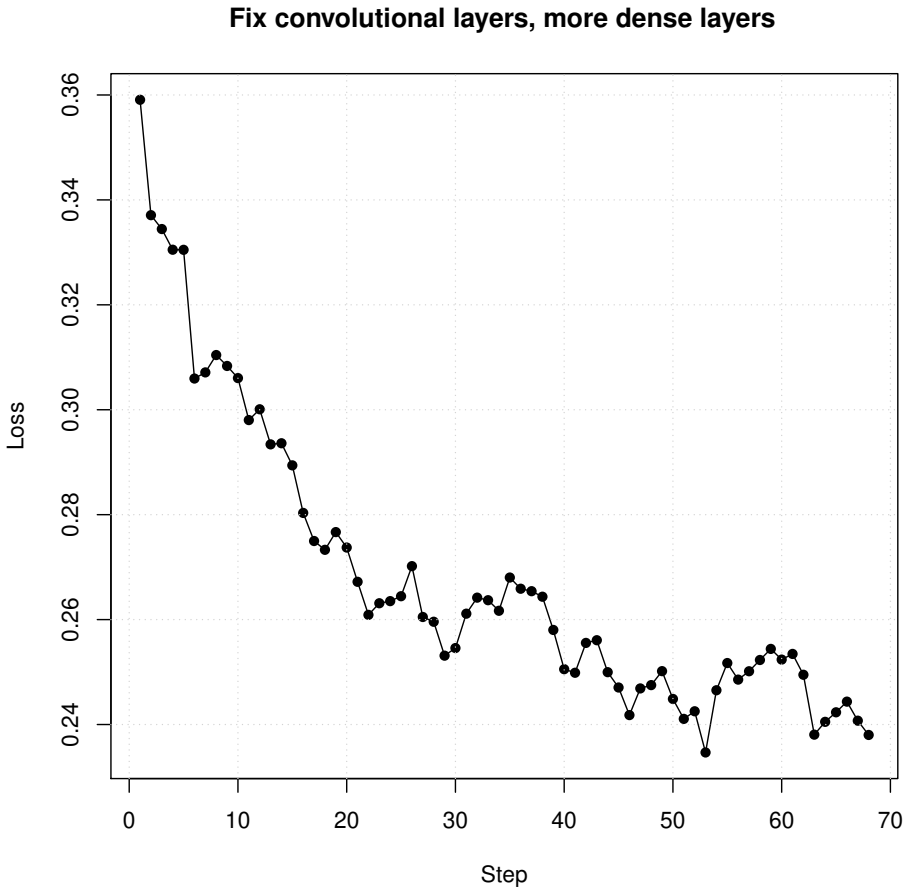


FIG. 6. This graph shows the cost function (or loss function) for the "more dense" model.

The cost function of this model shows that, indeed, it struggled during the training process. The cost value only got to 0.24 compared to the previous models.

4. Less convolutional layers, fix dense layers

This second to last model had less convolutional layers with respect to the base model. It also trained for 2 weeks. The following image shows the evolution of its accuracy:

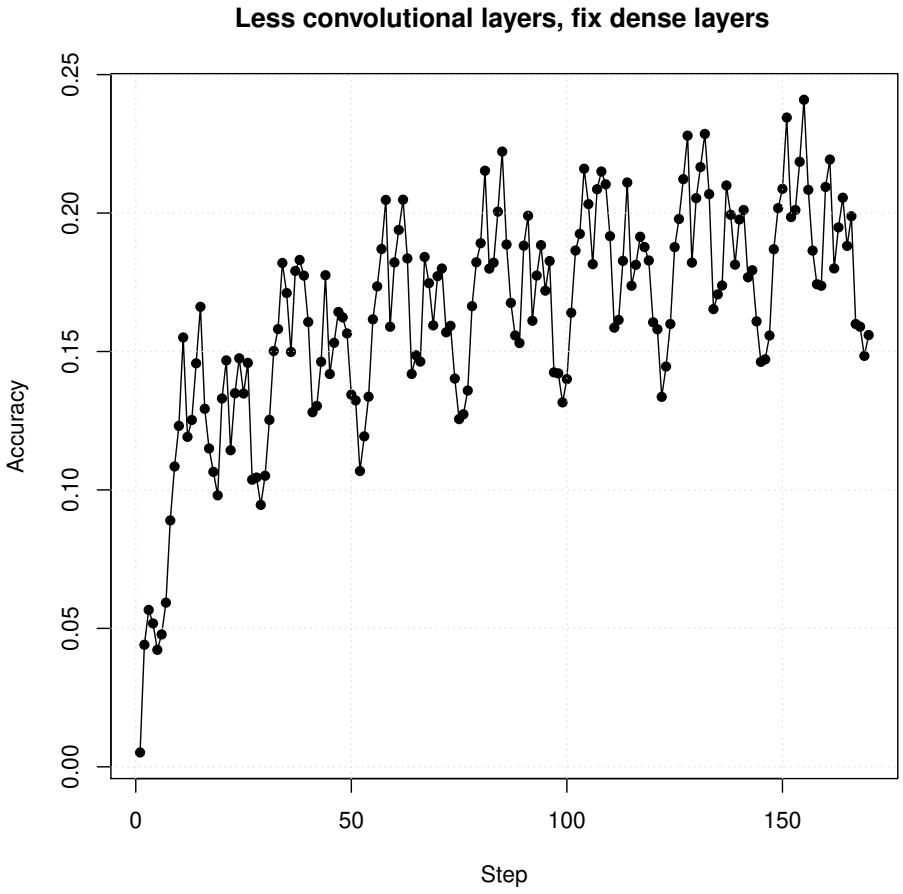


FIG. 7. This graph shows the accuracy of the "less convolutional layers" model.

Because this model had less convolutional layers, the total amount of parameters to be trained was $16,455,072$. This was reflected in the fact that this model trained for more than 150 steps.

Its accuracy surpassed the 20% mark, but didn't get to the 25% mark. The following graph shows its cost function:

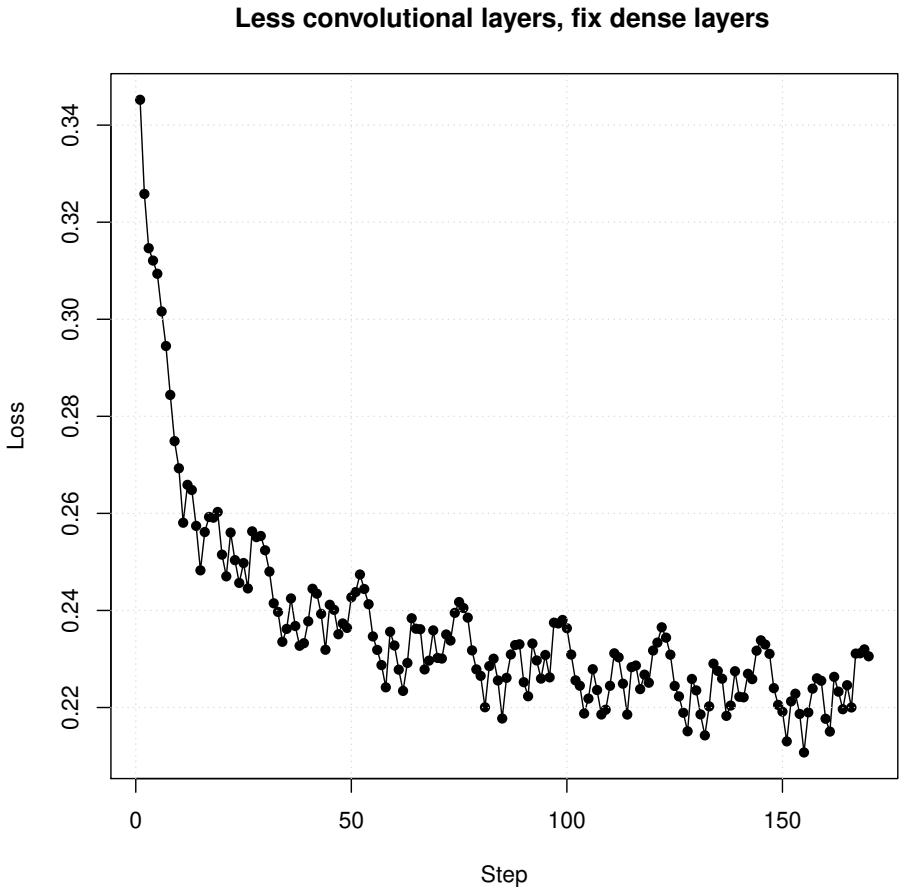


FIG. 8. This graph shows the cost function (or loss function) for the "less convolutional layers" model.

5. More convolutional layers, fix dense layers

Finally, the last model had way more convolutional layers compared to the base model. In particular, it had a total of 16 convolutional layers, which translated to a total amount of $112,802,592$ trainable parameters. This represented a huge downside during the training process, along with the fact that this model could only be trained for 2 days.

Due to these facts, the model could train for a total amount of only 14 steps. The results, though, were an accuracy of 0% and a cost value of 0.35 for the 14 steps. Although it's reasonable that the model could train for only 14 steps, it is a bit confusing the fact that didn't learn anything at all. This might be because of the huge amount of nudges that had to be applied to the weights and biases, but it could also be an error in the python code, during the process of building the network.

6. Overall conclusions for accuracy and cost function results

The following graphs show the accuracy and the cost function of all 4 models superposed (the last one isn't included because of the lack of data).

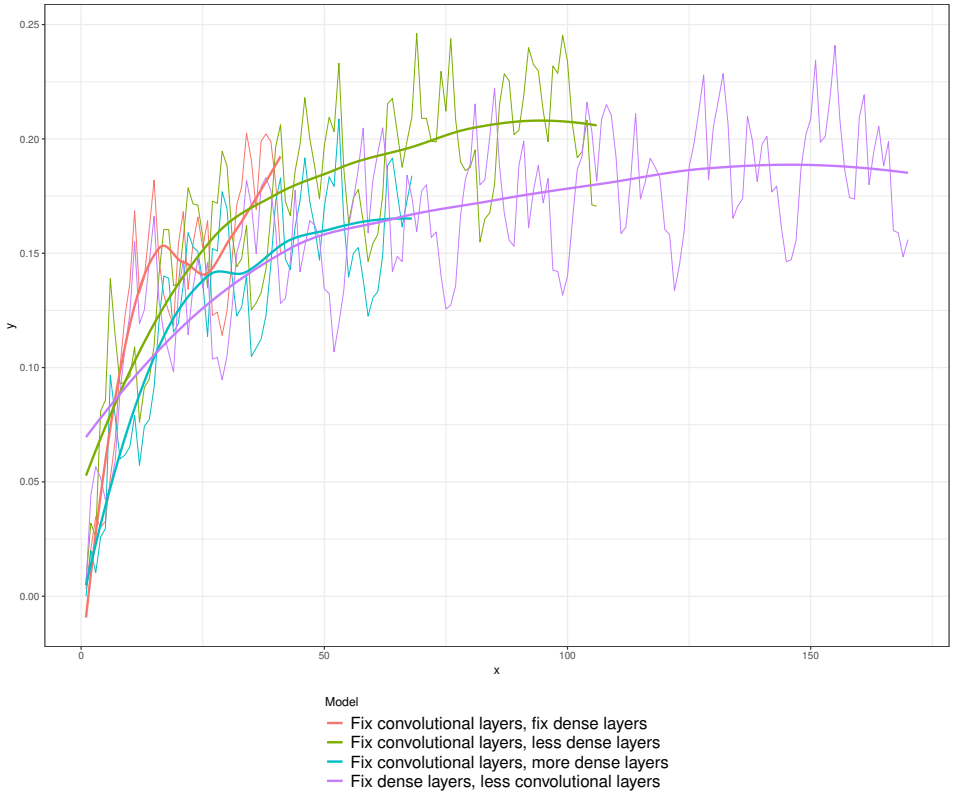


FIG. 9. This graph shows the accuracy of all models except for the "more convolutional layers".

From the previous graph, a couple of conclusions can be extracted:

- It seems as if increasing the number of dense layers (fully-connected layers) decreases performance, as the accuracy for the model "more dense" is lower than that from the "less dense" model. If the "less dense" model is instead compared to the base model, the accuracy is almost the same, although the base model was left training for only 2 days and it might have given better results if left trained for more time. This results can seem wrong at first, because more layers should translate to more learning capacity. Although this is true in most cases, it is indeed not true when both models had the same time to train. Probably, if more time had been given to the "more dense" model, the results would probably surpass the ones from the "less dense" model.

- If we focus on the convolutional layers, it seems that the base model performs better than the one with less convolutional layers. This makes sense, as chess is a game that relies a lot on figure patterns, which are recognized by convolutional layers. If we had results on the "more convolutional layers" model, this comparision could be made with more conviction.

The same conclusions can be extracted when looking at all cost functions superposed:

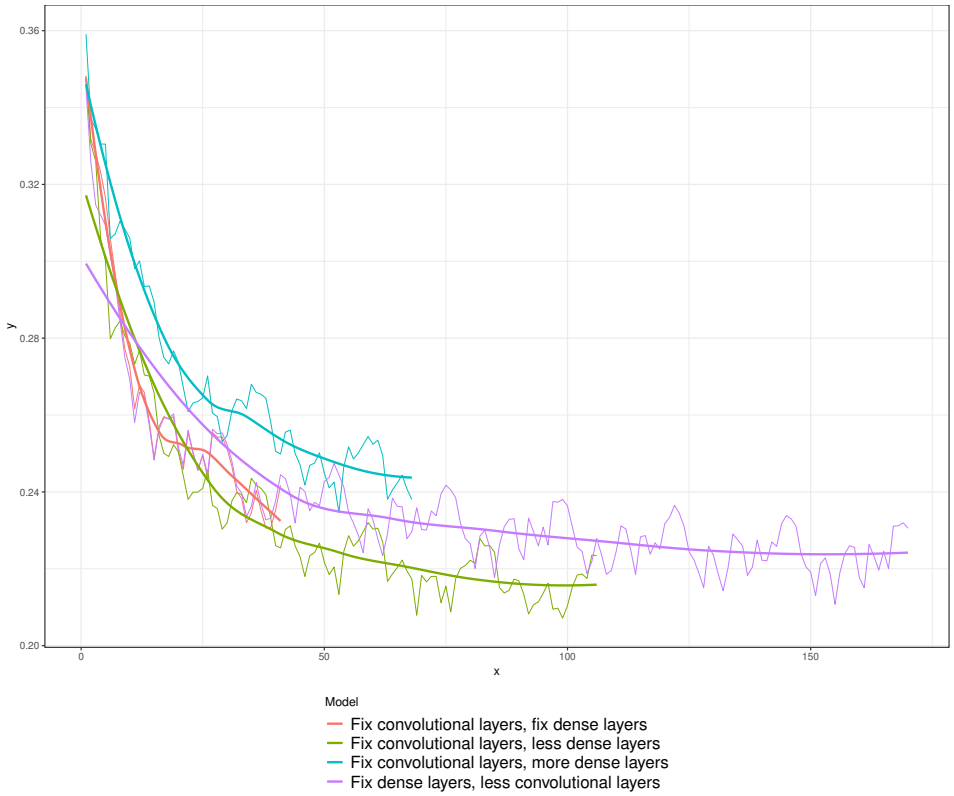


FIG. 10. This graph shows the cost function of all models except for the "more convolutional layers".

Chapter 11
R script results - Figure movement

In this second part of the "Results and Conclusions" chapter, the results from the R script will be discussed. Up until now, only the overall accuracy and cost function of the models have been compared. The accuracy tells how many predictions of the models correspond to the actual movements played by professionals on their games.

Thus, the accuracy is a pretty good approximation of the model's performance. But it only evaluates them using already played games by professionals. The accuracy doesn't tell if the models have really learned to play chess. In other words, the accuracy doesn't show if the model learned how figures move, or that figures should be moved, which where the initial goals for this project.

To evaluate how well the models learned the rules of chess, an R script was created in conjunction with a Python script, and a Bash script that encapsulates both. This scripts make use of the saved frames for each model during the training process. This way, a good overview of the learning process can be observed.

1. Python script

The Python script is the first one to be executed. Its function is to load random board positions from random games and ask each model to predict a move. This predictions are saved and later used by the R script.

To really see how each model learned over time, this script uses all frames saved during the learning process. Remember that each frame was saved once the cost value for a learning step reached a new lower value. Thus, in theory, each consecutive frame should represent a better version of the model. For each model frame it repeats the process of *asking the model to predict a move* for $10,000$ times. Before asking the model for a prediction, the random board position must be translated to the input data representation used by the model. This input data representation was presented in the "Data colllection and representation" chapter.

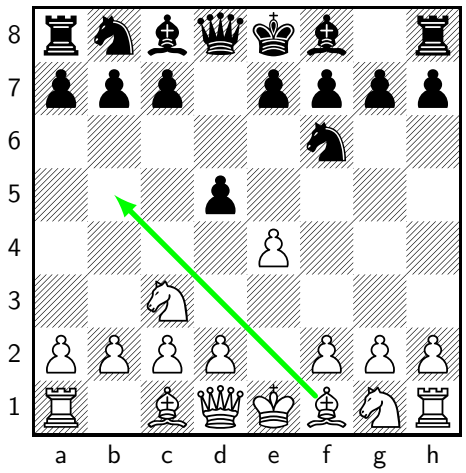
The predicted moves are saved in a dictionary data structure where the keys represent the figure and the values are a list of relative moves for that figure. The dictionary starts with empty lists for each figure:

```
{No_figure: [],
 Pawn: [],
 Rook: [],
 Knight: [],
 Bishop: [],
 Queen: [],
 King: []}
```

Notice that there are 7 entries in the dictionary, while there are 6 figures in chess. The first key is used to save the moves that a model predicts when no figure is present in the initial coordinates of the predicted move.

When the model predicts a movement, the initial coordinate is checked to see which figure the model is trying to move (or if no figure is present). Then, the relative vector is calculated and saved.

An example is now introduced to better understand this process. Imagine a loaded random board position that is fed to the network and the network predicts the move represented in a green line in the following diagram:



First, from the output returned by the model, the coordinates are extracted. Then, from the board position itself, the figure that is being moved is checked. In this case, Bishop. Then, the relative move is calculated. In this case the move is $(6, 1) \rightarrow (2, 5)$, which means that the relative move is: $(2 - 6, 5 - 1) = (-4, 4)$.

This relative move is then saved in the dictionary. In particular, the tuple $(-4, 4)$ is added to the list of the Bishop key from the dictionary[1]:

```
{No_figure: [],
 Pawn: [],
 Rook: [],
 Knight: [],
 Bishop: [(-4, 4)],
 Queen: [],
 King: []}
```

The following pseudocode shows the program flow of the Python script:

```
for each 'model_directory' in 'models_directory':
 for each 'model_frame' in 'model_directory':
  reset 'figure_dictionary'
  load 'model' from 'model_frame'
  for i in range(10000):
   load 'random_game'
   load 'random_position' from 'random_game'
   translate 'random_position' to 'input_representation'
   feed 'model' with 'input_representation'
   save 'predicted_move' to 'figure_dictionary'
  save 'figure_dictionary' to json file
```

Notice from the listing that once the $10,000$ predictions are computed for a particular frame, the dictionary is saved in a json file. Thus, if a model has 24 frames, it will end up with 24 json files with $10,000$ predictions each one.

The json file is then used by the R script to compute two statistics that will help visualize how well the model learned to play each figure.

2. R script

The R script is used to compute two statistics that will tell how well a model learned how chess figures moves. For this, it uses the json file generated by the Python script. The two statistics are:

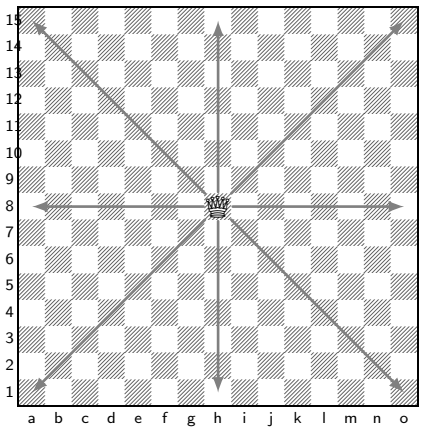
- Correct movements: This measure is the frequency sum of valid moves predicted by the model. For each model, each frame and each figure this measure is computed. For example, knowing that a Pawn can move to 4 positions depending on the board state, this measure will sum up the frequency of moves predicted for the 4 valid positions.

[1]In this example, the coordinate system used corresponds to that used universally in chess: the square $(1, 1)$ is positioned in the lower left corner. In the actual code, the coordinate system is changed so that the square $(1, 1)$ is positioned in the upper left corner. The reason for this was to facilitate coding and to ease computational cost.

- Correct destinations: This measure is the ratio between valid destinations and actual destinations predicted by the model. For example, a Pawn can move to 4 valid positions. If the model predicted a total of 14 different positions for Pawns (independently of the frequency at each one), this measure will be the ratio $4/14 = 0.28$ (assuming the 4 valid positions for Pawns are present in the 14 positions set predicted by the model).

The ideal value for these statistics would then be 1 (100%). For both statistics this would mean that all predictions were valid predictions, which means the model fully learned the movement patterns for each figure. It might be the case where the *Correct movements* statistic is very high but the *Correct destinations* statistic is very low, but not vice versa.

This statistics were calculated for each model, frame and figure. To better represent and visualize this values a basic schematic (or graph) was built using R. This schematic was generated for each frame and it contained both statistics as well as a graphic representation of the movement predictions generated by that model's frame. This graphic representation consists of a 15 by 15 squares grid. Each square is a possible destination for a movement starting at the center of the grid. Because the chess board size is 8×8, a valid move vector can go from -7 to 7, thus generating the 15×15 grid. The following chess board helps understand this graphic representation.



This graph representation for queen shows the valid directions for the queen to move. Starting at center $(0, 0)$ it can go to 8 directions ([(-7, -7), (-7, 0), (-7, 7), (0, 7), (7, 7), (7, 0), (7, -7), (0, -7)]).

In the graph representation generated by the R script, each square contains the frequency of predictions for that square. Also, the valid squares for the figure are highlighted to better visualize the predictions. The following image shows an example of a model frame for the Knight figure:

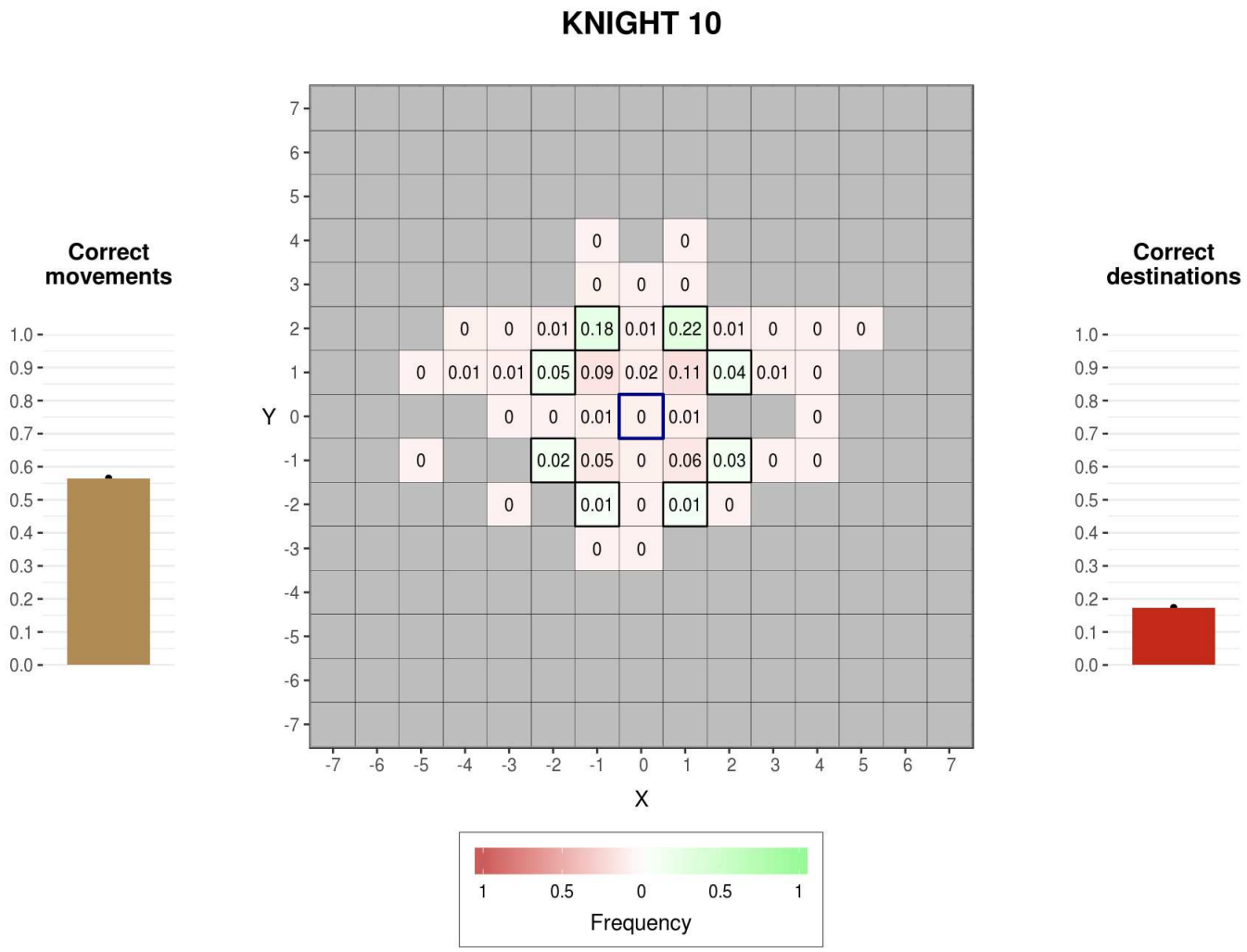


FIG. 1. This figure shows the graph representation generated by R for the Knight figure and frame number 10 (title). This particular example corresponds to the base model. The initial square is highlighted with blue, and will always be the center of the grid. The valid moves for a Knight are highlighted with wider black borders. The numbers inside the squares represent the frequency of predictions for a knight to that position. In red, the invalid positions, in green, the valid ones. Remember that each frame was tested with $10,000$ moves. The left bar shows the first statistic (Correct movements). Notice how the value corresponds to the sum of frequencies for the valid positions $(0.18 + 0.22 + 0.05 + 0.04 + 0.02 + 0.03 + 0.01 + 0.01 = 0.56)$. The right bar shows the second statistic (Correct destinations) which corresponds to the ratio of $8/46 = 0.17$. These schematics can be found in the Appendix of this thesis.

Finally, because the 10,000 were randomly chosen, the distribution of every chess figure picked by the model will be different. In other words, the model might be biased to always pick the Queen, for example. This information is very important when evaluating the graph representations shown above. Because this information is not present in these schematics, another graph was created for every model's frame, indicating the distribution of figure picks. The "no-figure" figure was also included in this graph to see how many times the model tried to move a "no-figure" with respect to the rest of figures.

3. Fix convolutional layers, fix dense layers (base model)

Again, this is the base model from which the other models were built for comparision purposes. As explained in the Tensorboard results this model run for only 2 days. During that process, 22 snapshots of the model were saved. This means that there were 22 times the cost function of the model reached a new lower value. Because the graph representations containing the 15×15 grid are to big and there are too many (22 for every chess figure in this case), they are shown in the Appendix of this project. In this section only the two statistics and the piece distribution will be discussed.

The following graph shows the Correct movements statistic for all 6 figures for the base model:

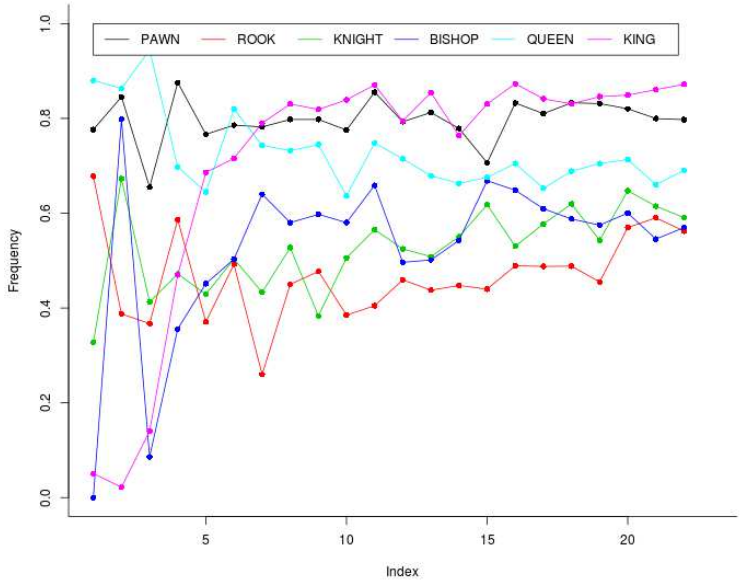


FIG. 2. This graph shows the correct movements statistic for the base model.

With this graph, it becomes very clear how well the model learned to play different figures. Notice that the frequency for all figures is greater than 0.5. This means that whenever the model tried to move a figure, more than 50% of the times it moved it correctly. Considering that there are a total of $8 \times 8 = 64$ squares in a chess board and that a figure can move to a maximum of 21 squares[2] the results are quite good for a learning period of just 2 days.

When comparing figures, other conclusions can be extracted. Notice how those figures with less valid destinations (for example, King and Pawns) have higher frequencies. This makes sense, because the model is capable of learning "small" patterns more quickly (the neurons that fire when a Pawn is moving are much more closer together). In the other hand, figures that have long ranges to go to have a lower frequency (Queen, Bishop, Rook). Although Queen have the highest amount of possible destinations, it is not the one with less frequency. This is because when trying to move the Queen, the probability of ending up in a valid destination is higher, despite the move being wrong in the current board context. Finally, the Knights also have a lower frequency, but with less valid destinations to go to. Why are they so difficult to learn then? A possible explanation resides in the network structure. The positions to where Knights can go are few but sparse, which means that the neurons that fire when trying to move a Knight are more distant from each other, which may difficult the learning process for this figure.

Another interesting graph to look at is the distribution of figure picks for the $10,000$ predictions. The following two images shows this distribution for the first snapshot or frame (before any training) and the last one (fully trained for 2 days):

[2]the queen, when all its moving axes are clear, can move to 21 different positions, which is the maximum among all figures.

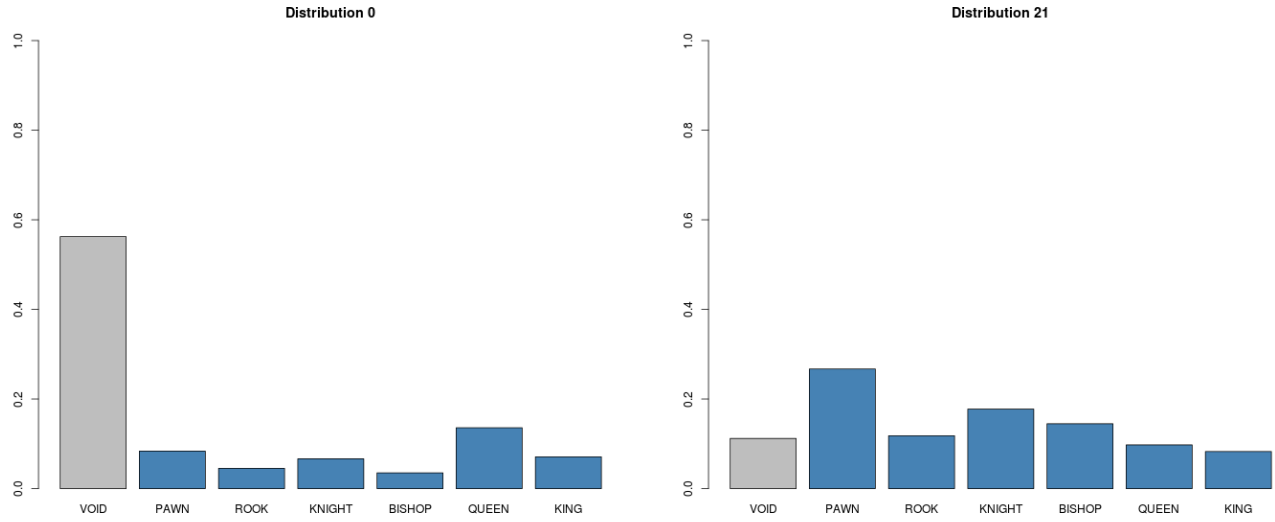


FIG. 3. Figure picks distribution for first snapshot (left) and last snapshot (right) for base model.

This histograms show that not only the model learned how each figure moves, but also that they have to be moved. Almost 50% of the time the model missed a figure in the first snapshot (before training). In the last snapshot, with only 2 days of training, only 10% of the moves were a miss. Knowing that each frame correspond to a total of $10,000$ moves, this means that initially the model missed approximately $5,000$ moves, whereas after training it only missed $1,000$.

Another interesting thing to do with this results is to compare the discrete probability distribution of figure picks just shown with a real probability distribution from professional players. The following graph shows the probability distribution of figure picks in professional games.

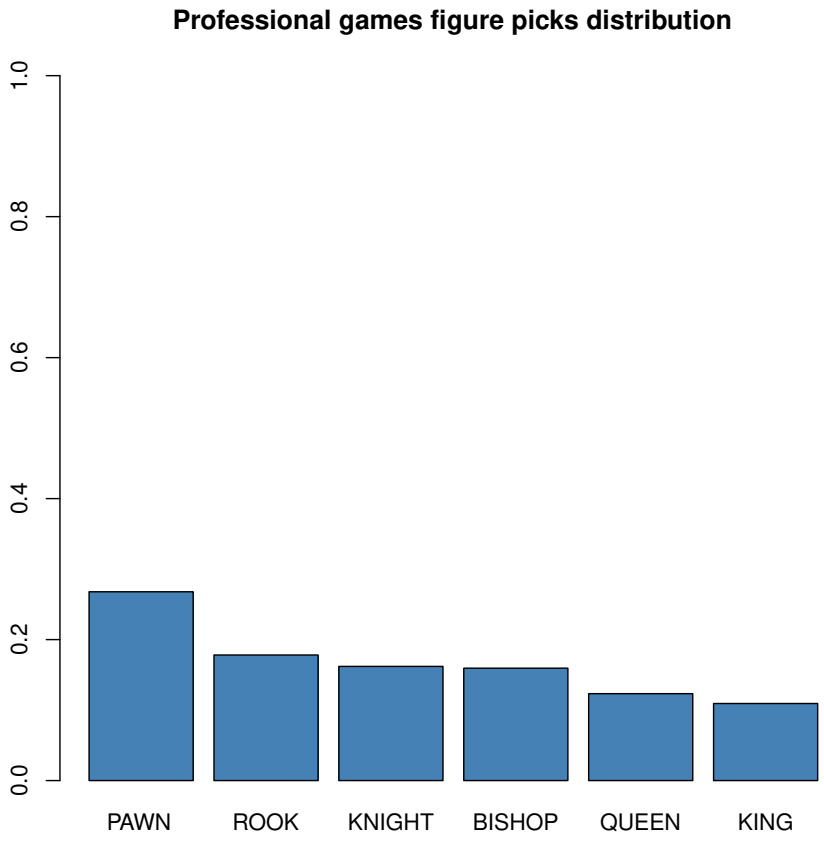


FIG. 4. This graph shows the probability distribution of chess figure picks in professional games. The data was collected by a software developer at Profound. The data was collected from a total of more than 4 million professional games [**22**].

As it can be quickly seen from the image above, the final figure picks distribution of our model and the figure picks distribution of professional players is quite similar, except for Rook and Knight figures. This is another good indicator that the model actually learned what figure to move depending on the board state.

The following graph shows the second statistic (Correct destinations):

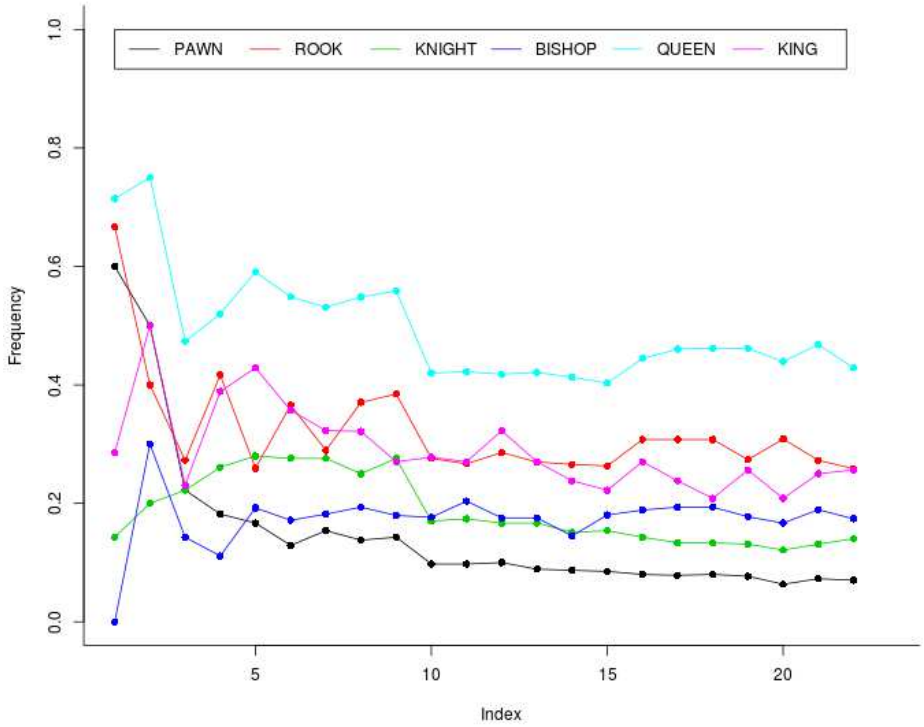


FIG. 5. This graph shows the correct destinations statistic for the base model.

At first glance, this graph may give the impression that the model didn't actually learn how figures move. Remember that this statistic represents the number of valid final positions predicted for a figure divided by the total amount of different predicted final positions. This means that for Pawn, after 2 days of learning only 10% of predicted final positions where a valid destination for Pawn. Not only this can be discouraging, but also the fact that the graph shows a negative trend, which might be understood as the model forgetting rather than learning.

Although this is true when the Correct destinations graph is observed, the overall graph illustration generated by R and presented before shows a more convincing result. The graph illustration for figure Pawn at last snapshot is shown:

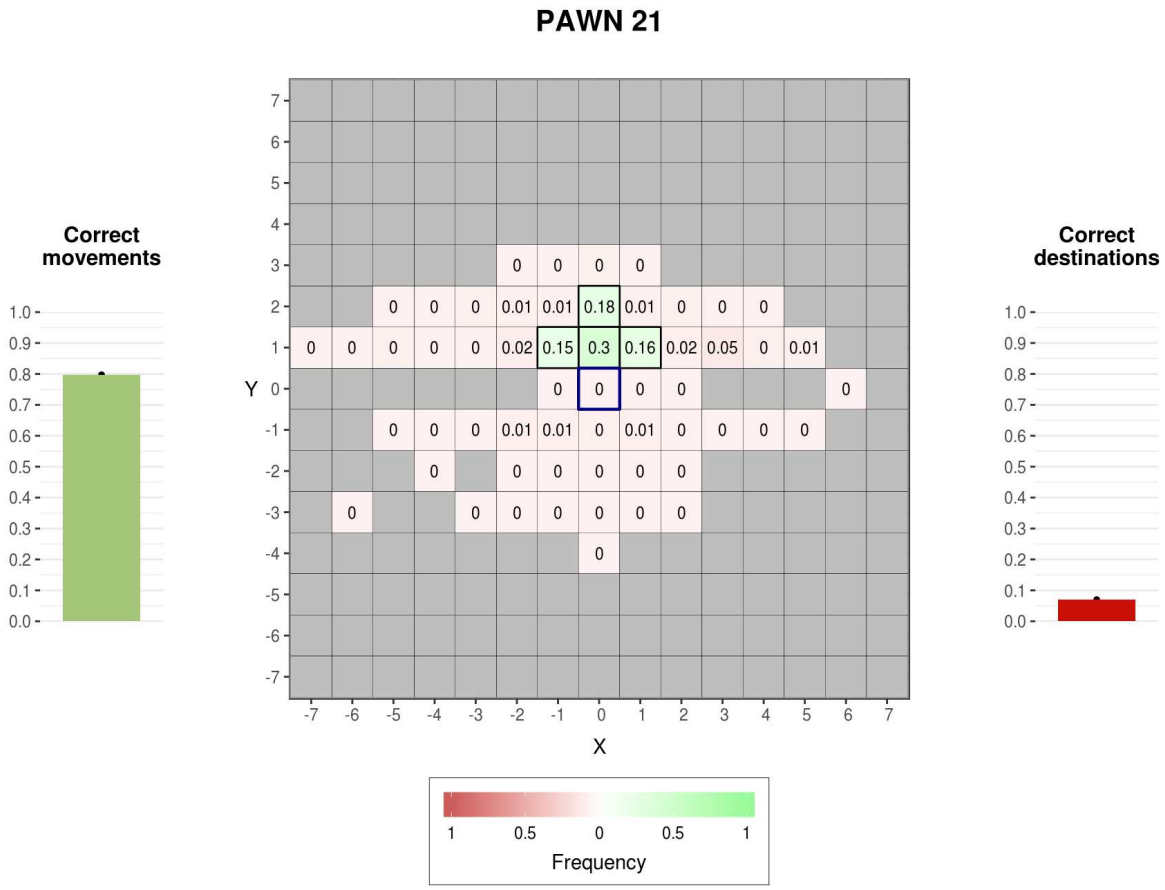


FIG. 6. This graph shows the overall graph illustration for Pawn at snapshot 21 and base model.

With this illustration it becomes clear what is going on and why the Correct destinations statistic is so low. Notice that all red squares are invalid destinations, but the sum of those squares are only approximately 20%! In other words, the model actually learned how Pawn moved, but from time to time it makes an erroneous movement. This erroneous movement impacts greatly to the Correct destinations statistic, but the Correct movements statistic shows that the model is actually learning.

But why is there a down trend from the first snapshot to the last one? A logical explanation is that initially the model doesn't know what is doing, at all. It randomly choses initial and final positions. In fact, because the weights and biases are not adjusted in the first snapshot, this random initial and final positions are

the same for all figures. During the training process, the model learns how big the board is and that figures might eventually move to more distant positions. For example Queens can move to the other side of the board in one move. While the model is learning all this, it is also learning how each figure moves. This means that in some cases, when trying to move a Pawn, it gives a final destination that is not valid for Pawns. The knowledge of how the model learned during the training process makes this erroneous predictions understandable. In fact, notice how low the frequency is for most of them: less than 0.005 knowing that the number is rounded to 2 decimals.

Also from 5, when looking at individual figures, a pattern can be observed. Those figures with less valid destination positions have a lower value for the Correct destinations statistic. In the other hand, the figures with more valid destination positions have a higher value for this statistic. This makes sense, because for queen, more valid destination positions exist and the probability of making a move to one of them is greater. Thus, despite the prediction destination being very bad considering the board state, it will be valid nonetheless.

For the rest of the models, the same individual conclusions can be extracted. Thus, in the following sections only the graphs will be shown. A final overall conclusion will come right after.

4. Fix convolutional layers, less dense layers

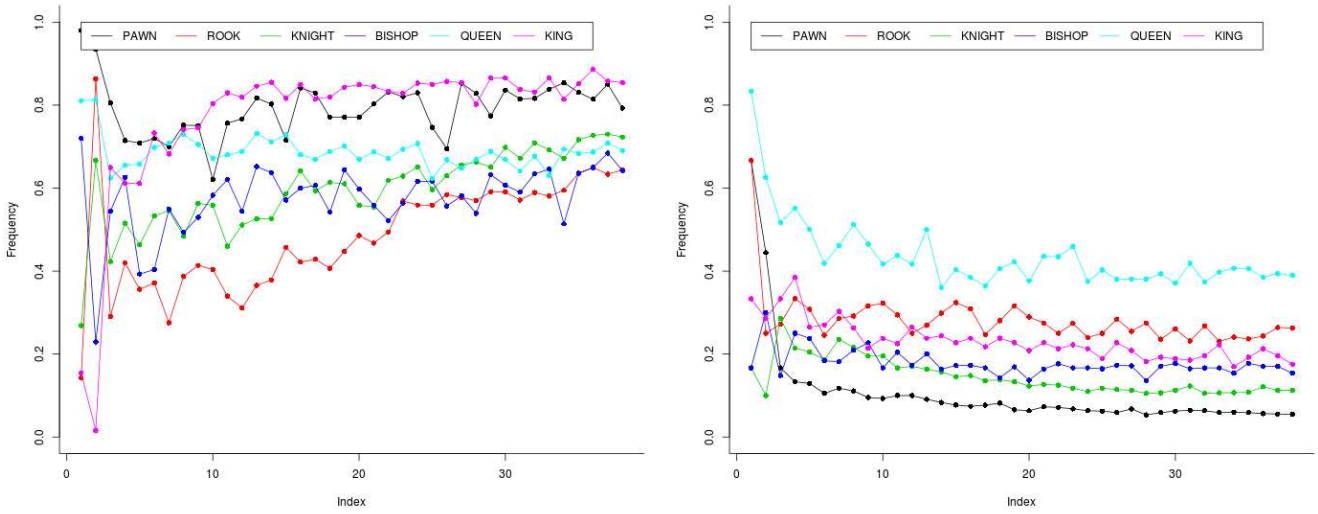


FIG. 7. Correct movements (left) and correct destinations (right) for less dense model.

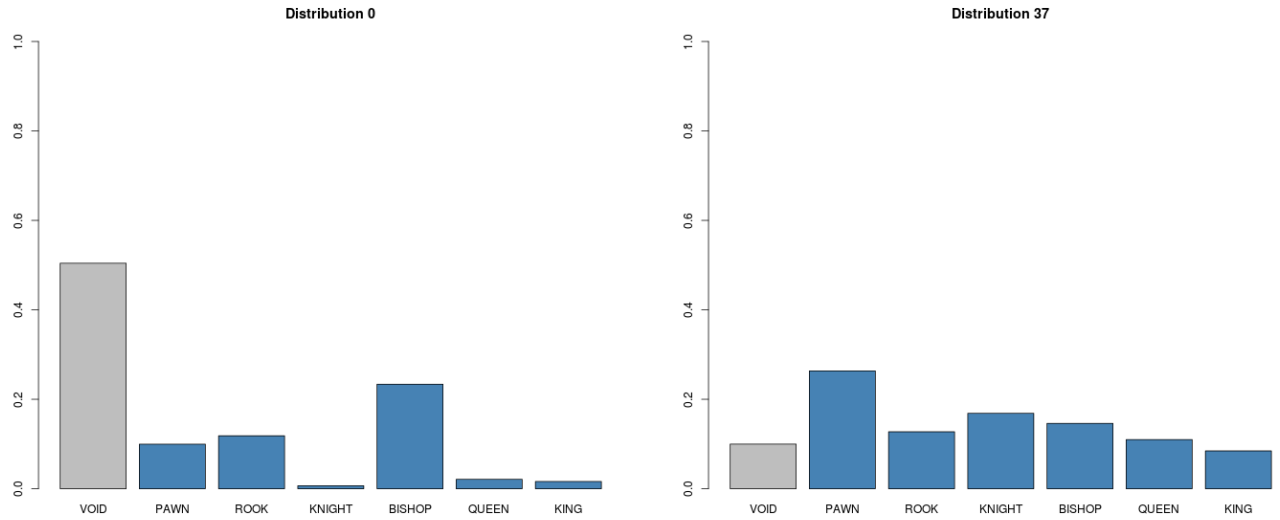


FIG. 8. Figure picks distribution for first snapshot (left) and last snapshot (right) for less dense model.

5. Fix convolutional layers, more dense layers

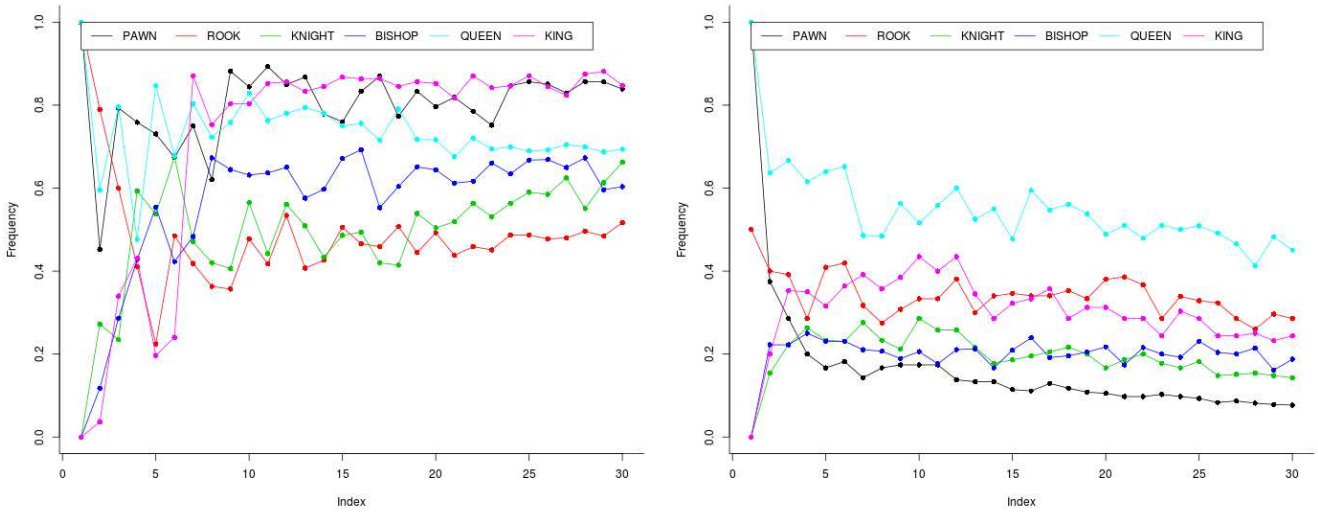


FIG. 9. Correct movements (left) and correct destinations (right) for more dense model.

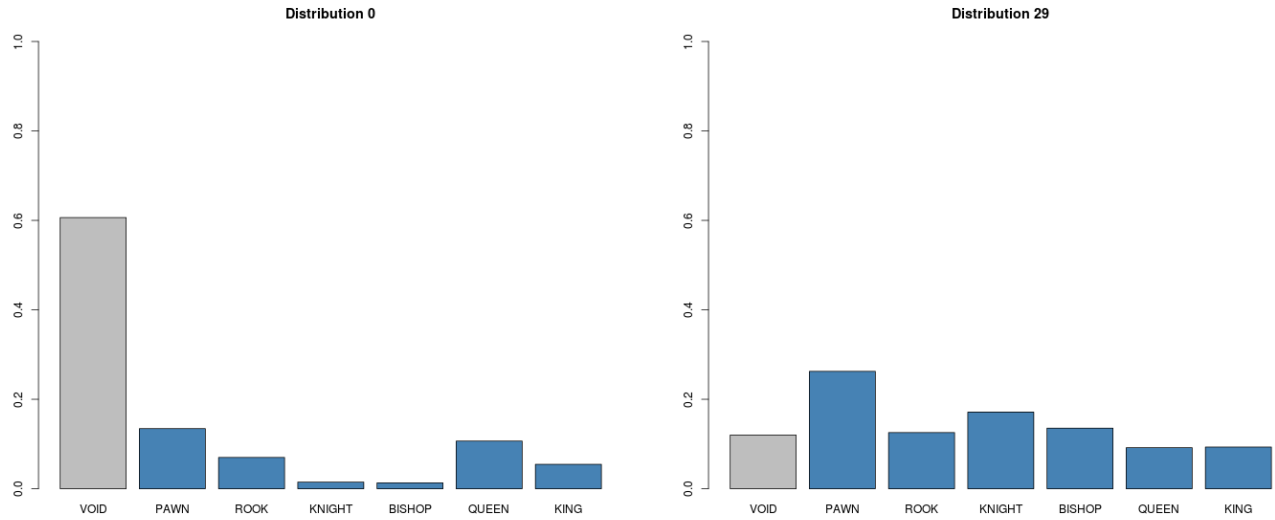


FIG. 10. Figure picks distribution for first snapshot (left) and last snapshot (right) for more dense model.

6. Less convolutional layers, fix dense layers

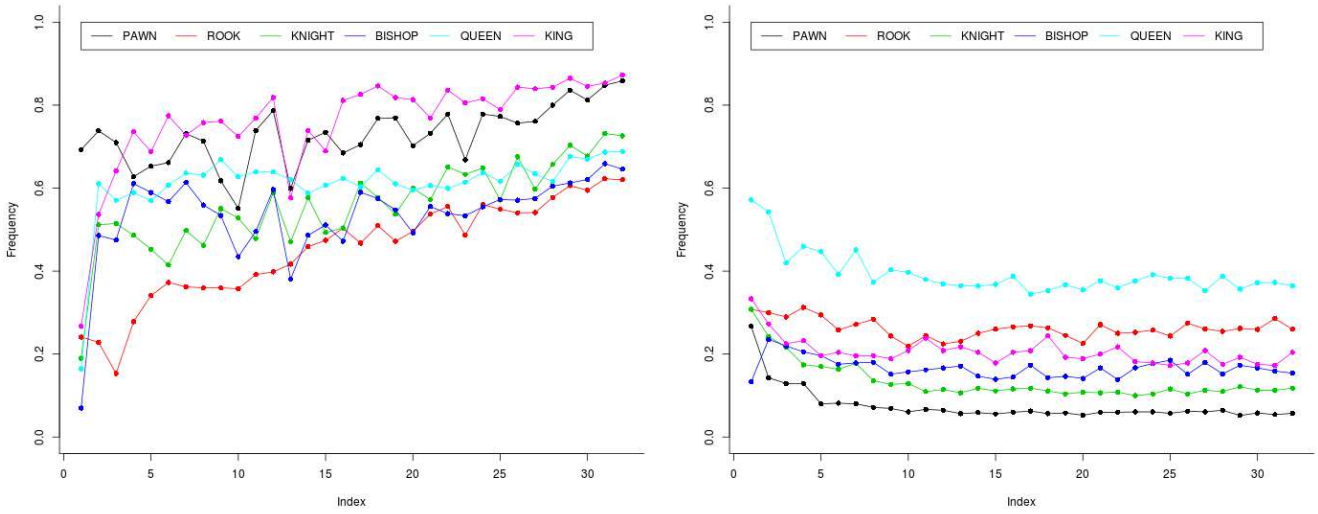


FIG. 11. Correct movements (left) and correct destinations (right) for less convolutional layers model.

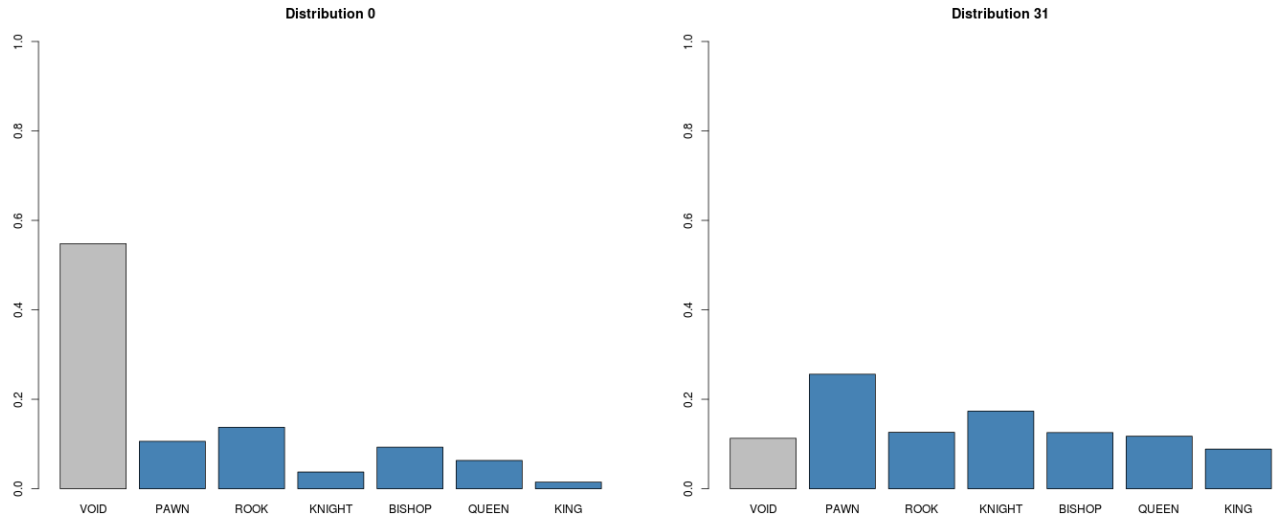


FIG. 12. Figure picks distribution for first snapshot (left) and last snapshot (right) for less dense model.

7. More convolutional layers, fix dense layers

As explained in the previous chapters, this model could only be trained for 2 days. And because of its number of parameters to be trained it only could train for 14 steps with very poor results. Thus, this model is not added in this comparision.

8. Overall conclusion for R Script results

As previously mentioned, the conclusions extracted individually from each model are the same and are explained in the base model. In this section, the overall conclusions will be presented.

- First of all, looking at the figure picks distribution for all models, it is clear that all learned in the same way that figures are the ones to be moved. Also, the distribution for all figures is quite similar for all models, so no conclusions can be extracted from this.
- Secondly, the Correct destinations graphs for all models are quite similar too. The down trend is visible in all models and the order of the lines for each figure is the same. Again, no conclusion can be extracted from this to compare the different models.
- Finally, looking at the Correct movements graph, the differences between models appear. The "less dense" model seems to have better frequencies for all figures, specially for the ones with larger ranges of movement. In contrast, the "more dense" model show worse results compared to both the base model and the "less dense" model. This conclusions are, in fact, the same derived from the accuracy and cost function results seen in the last chapter. In other words, the results from the R script correlate with the ones obtained from Tensorboard. Looking at the "less convolutional layers" model, the results are quite similar to the base model. This is contradictory to the results obtained in Tensorboard, where it seemed that less convolutional layers correlated with lower accuracy. A plausible explanation for this is that "less convolutional layers" model was able to learn how to move each figure, but wasn't able to learn how to use them.

From both the results presented in this chapter and the ones presented in the previous chapter, it seems clear that "less dense" model was the model with better results. In one hand, it had the best accuracy and lower cost function values. In the other hand, the model learned ho to move each figure better than the others.

Although it is true that "less dense" model seems to have outperformed the others, it might seem counterintuitive. A initial hypothesis would be that with more layers (both dense and convolutional) a model should be able to play better. With more convolutional layers it should be better recognizing more complex patterns in the board. With more dense layers, it should be able to extract more information from this patterns.

As previously pointed out, a possible explanation as to why these have not been the results seen in this project is that all models had the same time for training,

independently of the time required for each model to learn the same. Probably, if more time had been given to the "more convolutional layers" and "more dense layers" models, the results would have been different.

Another important thing to take into consideration is that not always more layers correlates with more accuracy. It is well known that very complex models tend to overfit the data, which could be another plausible explanation to why the other models didn't perform as well as the "less dense" model.

Chapter 12
Final thoughts and further work

The aim of this project was to understand in detail the inner workings of deep neural networks. In particular convolutional neural networks. Also, the development of different convolutional neural networks that learned to play chess from scratch was one of the main goals.

Although both goals have been completed, more further work could be used to improve the models created. First of all, the models created only differ from each other in 2 aspects: the number of convolutional layers and the number of dense layers. Although the 2 aspects studied are very important in a network structure, there are a lot more hyper-parameters that could be tested between models. The following list shows some of these other hyper-parameters:

- Number of neurons per convolutional layer
- Number of neurons per dense layer
- Activation function for each neuron
- Number of pooling layers
- Number of dropout layers
- Initial distribution for weights and biases
- Kernel sizes for the convolutional layers
- The use of padding in during the convolutions
- ...

This and many more small parameters could have been tuned when creating the neural network structures, but the results might have been the same in the end. The only thing that would have probably changed is the process of learning more than the final results.

Another thing that is important to keep in mind is the fact that no cross-validation technique was used during the learning process. This can be seen when looking at the accuracy graphs for every model. A clear pattern appears in all graphs which indicates periodicity in the learning process. The problem with not using cross-validation techniques is that the model tend to overfit the data. Cross-validation techniques solve this problem by removing part of the training dataset while learning and using it for testing. Although its true that cross-validation could have improved the learning process, the cost function for the validation dataset for all

models doesn't show any overfit at all. When a model overfits the training data, the cost function for the validation dataset creates a 'U' type graph. That's not the case in results presented.

The neural networks trained in this project have learned the basics of chess. That is, that chess figures should move and how should they move. But no further improvement was noticed in the results. In other words, the models learned the basic patterns of chess looking at how professional players played, but they didn't build any strategy. This becomes clear when trying to play against one of the models. If the human player follows correctly a well-known chess opening, the model will be able to follow up. But if for testing purposes the human unprotects its Queen with a stupid move, the model will probably not kill it. The models have learned to repeat the professional plays, but didn't build any strategy.

To further improve this, other types of machine learning techniques can be used. For example, the use of reinforcement learning in chess machines have been very successful. Using reinforcement learning, the models learn by playing against themselves. This way, the models learn new patterns and strategies in order to win, rather than repeating professional moves.

Another interesting thing would be to create a visual representation of the trained kernels for each model. Often this visual representation helps to understand how the network thinks and what patterns it recognizes from the board position.

A nice thing about using convolutional neural networks with board games is that it is relatively easy to train and understand what the network is doing. In chess, it is easy to see what figure the model is trying to move and to understand why it's doing so. But the full potential of convolutional neural networks reside in other disciplines, like research or medicine. Once the inner workings of these networks are understood when applied in board games, they can be used in these other disciplines with very small changes.

References

[1] Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), pp.484-489.

[2] B2SLab. (2018). B2SLab — Bioinformatics and Biomedical Signals Laboratory. [online] Available at: http://b2slab.upc.edu [Accessed 27 Sep. 2018].

[3] GitHub. (2018). keras-team/keras. [online] Available at: https://github.com/keras-team/keras [Accessed 27 Sep. 2018].

[4] TensorFlow. (2018). TensorFlow. [online] Available at: https://www.tensorflow.org [Accessed 27 Sep. 2018].

[5] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), pp.386-408.

[6] Adesola (2015). Artificial Neuron Network Implementation of Boolean Logic Gates by Perceptron and Threshold Element as Neuron Output Function.

[7] Nielsen, M. (2018). Neural Networks and Deep Learning. [online] Neuralnetworksanddeeplearning.com. Available at: http://neuralnetworksanddeeplearning.com [Accessed 27 Sep. 2018].

[8] Jobran Al-Mahasneh, A., G. Anavatti, S. and A. Garratt, M. (2018). Review of Applications of Generalized Regression Neural Networks in Identification and Control of Dynamic Systems. School of Engineering and Information Technology.

[9] Rocha, A. and Klein Goldenstein, S. (2014). Multiclass From Binary: Expanding One-Versus-All, One-Versus-One and ECOC-Based Approaches. IEEE Transactions on Neural Networks and Learning Systems, 25(2), pp.289-302.

[10] Lipton, Zachary. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning.

[11] Hochreiter, S. & Scmidhuber, Ju. (1997), 'Long short-term memory', Neural computation 9 (8), 1735–1780.

[12] Cybenko, G. (1992). Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems, 5(4), pp.455-45

[13] Deeplearningbook.org. (2018). Deep Learning. [online] Available at: http://www.deeplearningbook.org [Accessed 27 Sep. 2018].

[14] Hubel, D. and Wiesel, T. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. The Journal of Physiology, 160(1), pp.106-154.

[15] N. P. Jouppi, *et al* (2017) In-datacenter performance analysis of a tensor processing unit. 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)

[16] Saremba.de. (2018). Standard: Portable Game Notation Specification and Implementation Guide. [online] Available at: http://www.saremba.de/chessgml/standards/pgn/pgn-complete.htm [Accessed 28 Sep. 2018].

[17] Oshri, B. (2015). Predicting Moves in Chess using Convolutional Neural Networks.

[18] Chess-db.com. (2018). Chess Collections for Download. [online] Available at: https://chess-db.com/public/downloads/gamesfordownload.jsp [Accessed 27 Sep. 2018].

[19] KingBase. (2018). KingBase - a free chess games database, updated monthly.. [online] Available at: http://www.kingbase-chess.net [Accessed 27 Sep. 2018].

[20] (2018). What is the average length of a game of chess?. [online] Chess Stack Exchange. Available at: https://chess.stackexchange.com/questions/2506/what-is-the-average-length-of-a-game-of-chess [Accessed 27 Sep. 2018].

[21] en.wikipedia.org. (2018). Promotion (chess). [online] Available at: https://en.wikipedia.org/wiki/Promotion_(chess) [Accessed 27 Sep. 2018].

[22] Vanheusden.com. (2018). Data-mining on the game of chess. [online] Available at: https://www.vanheusden.com/cchess/datamining_on_chess.html [Accessed 27 Sep. 2018].

Part 4

Appendix

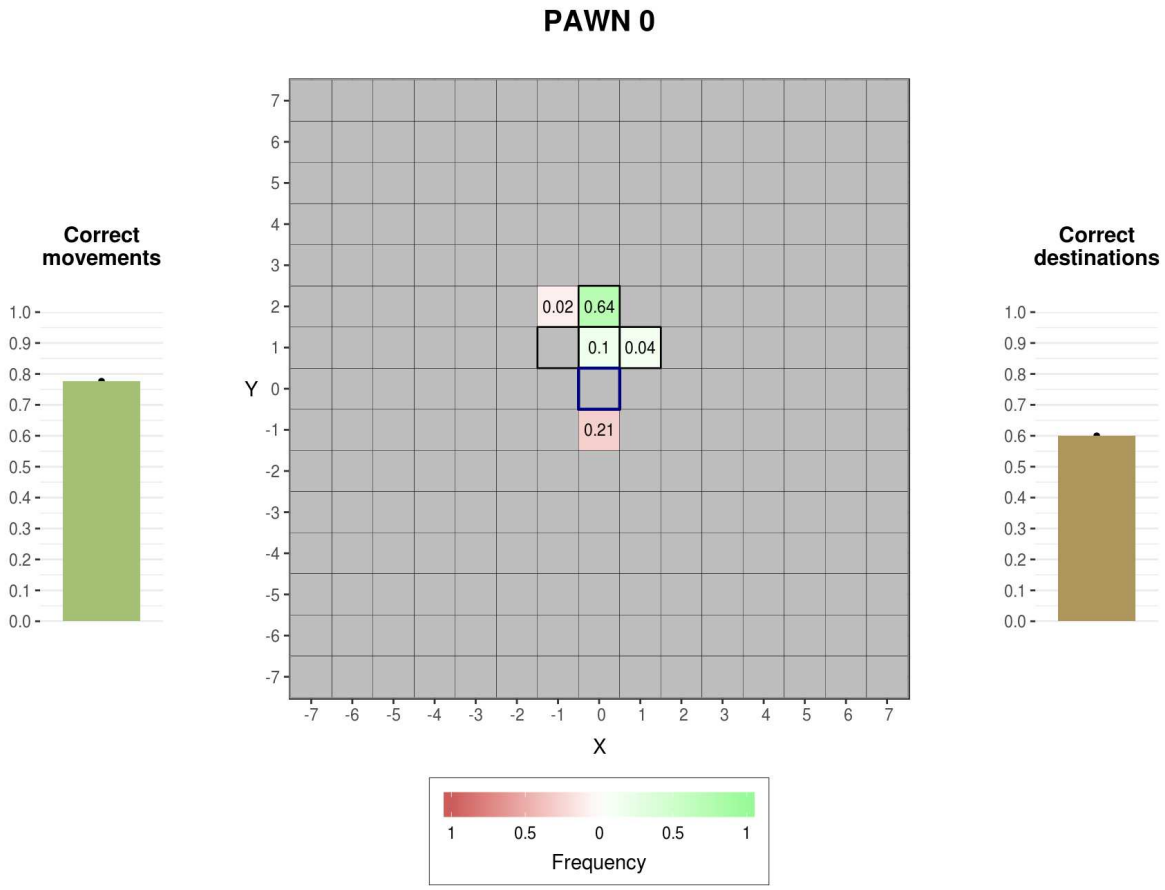


FIG. 1. This graph shows the R script schematic for Pawn at snapshot 0 and base model.

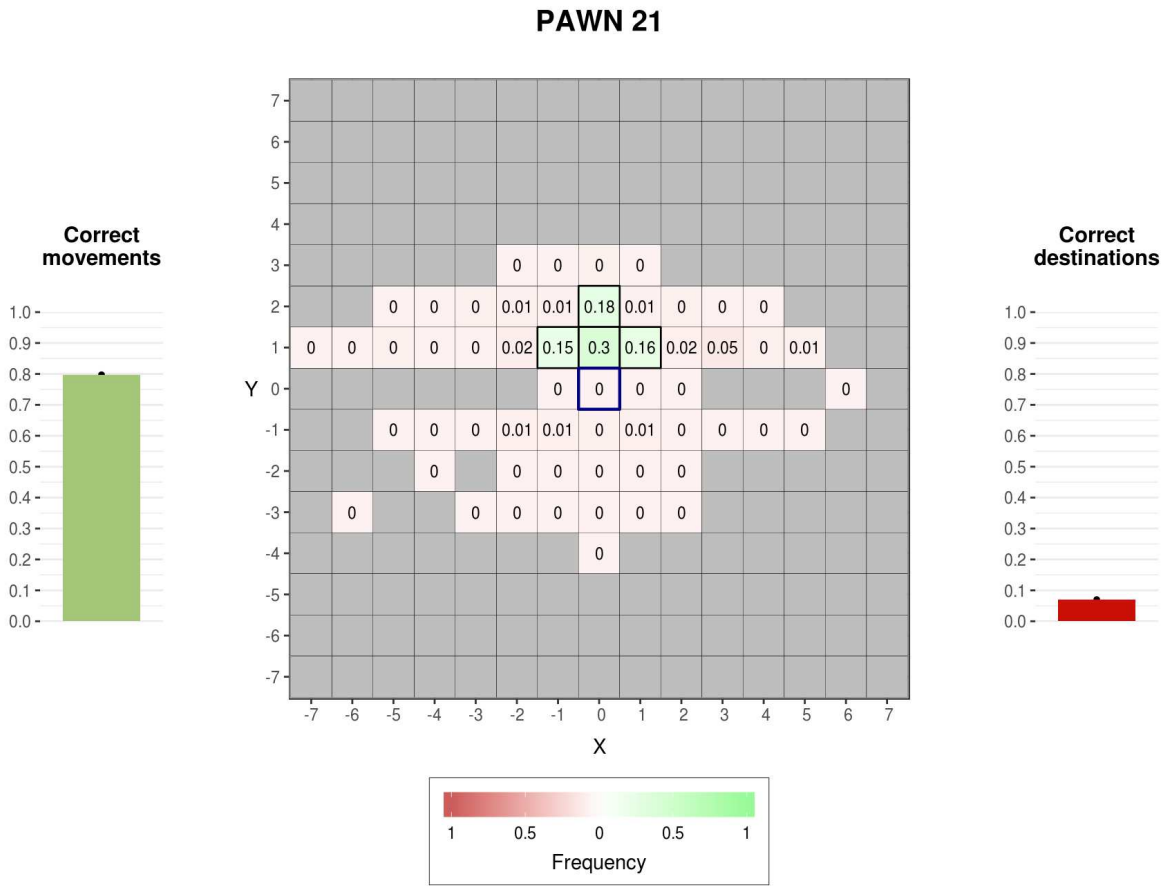


FIG. 2. This graph shows the R script schematic for Pawn at snapshot 21 and base model.

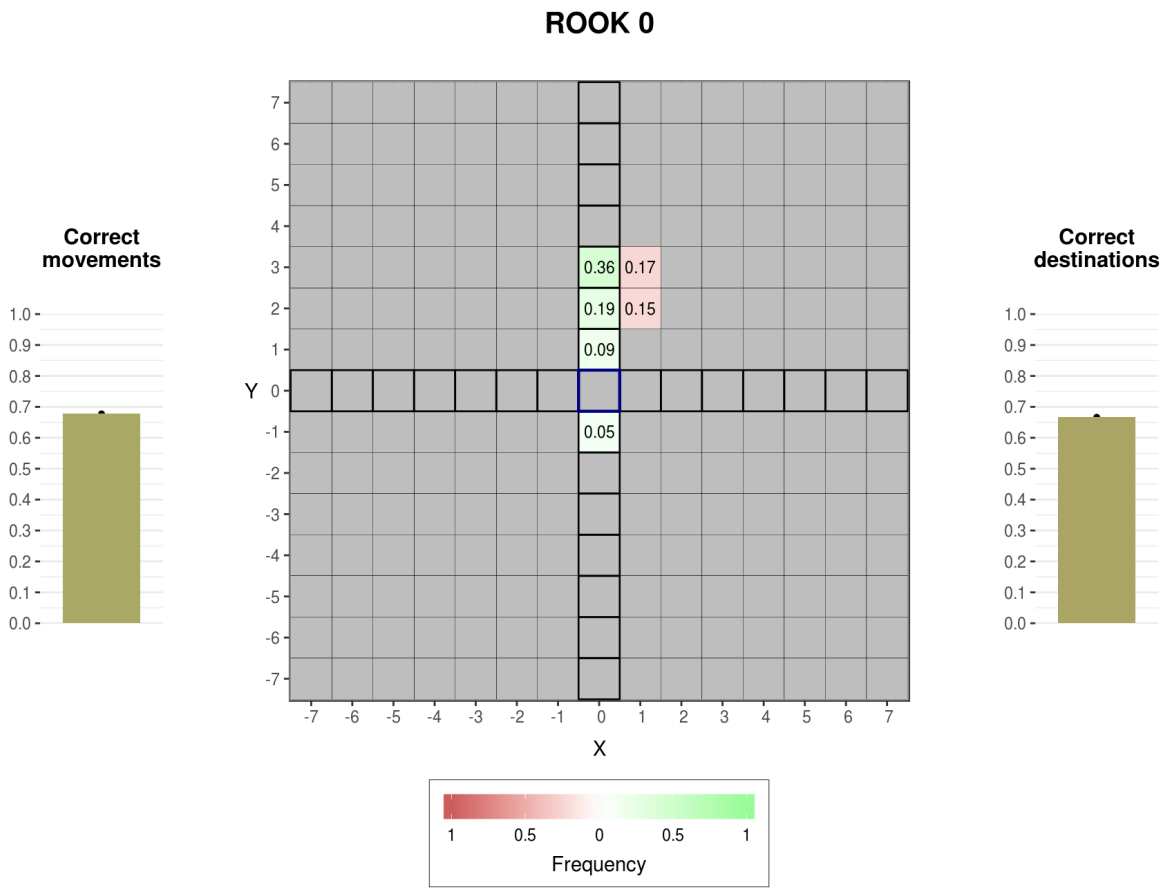


FIG. 3. This graph shows the R script schematic for Rook at snapshot 0 and base model.

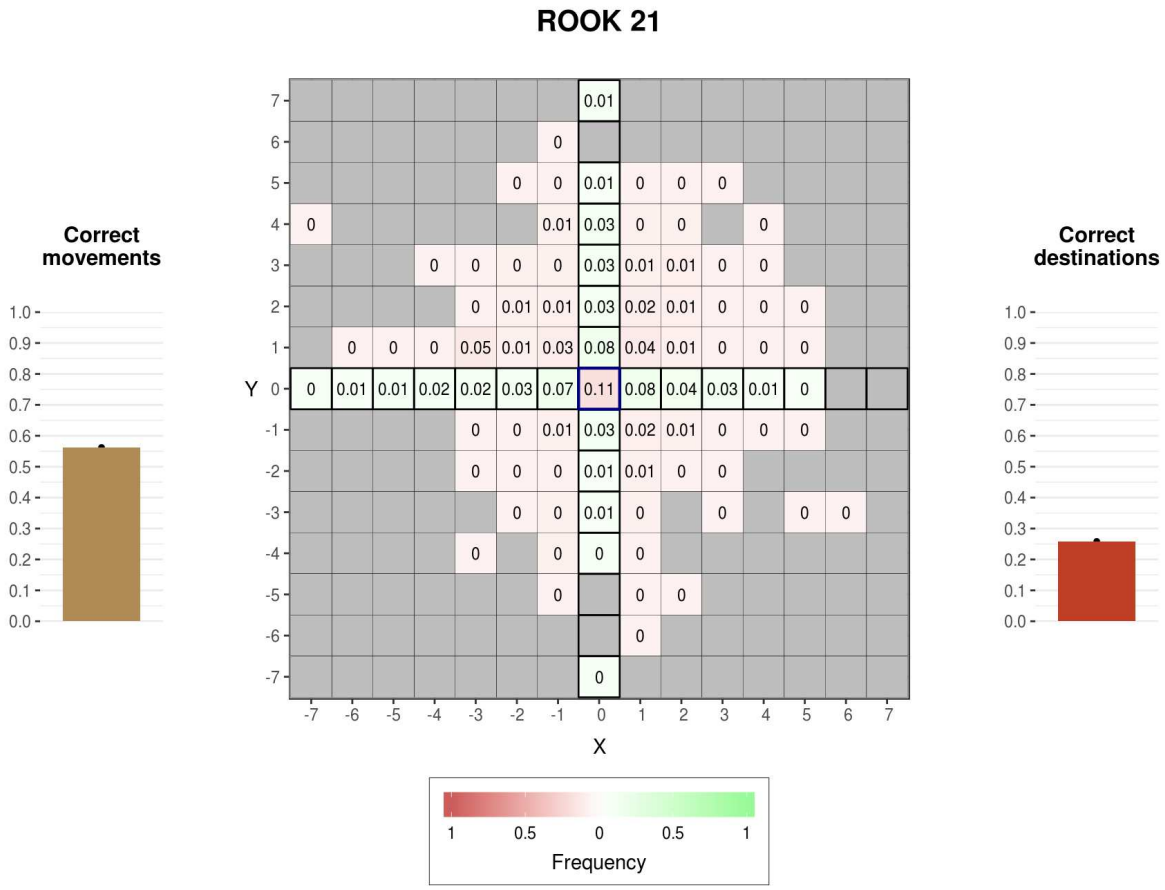


FIG. 4. This graph shows the R script schematic for Rook at snapshot 21 and base model.

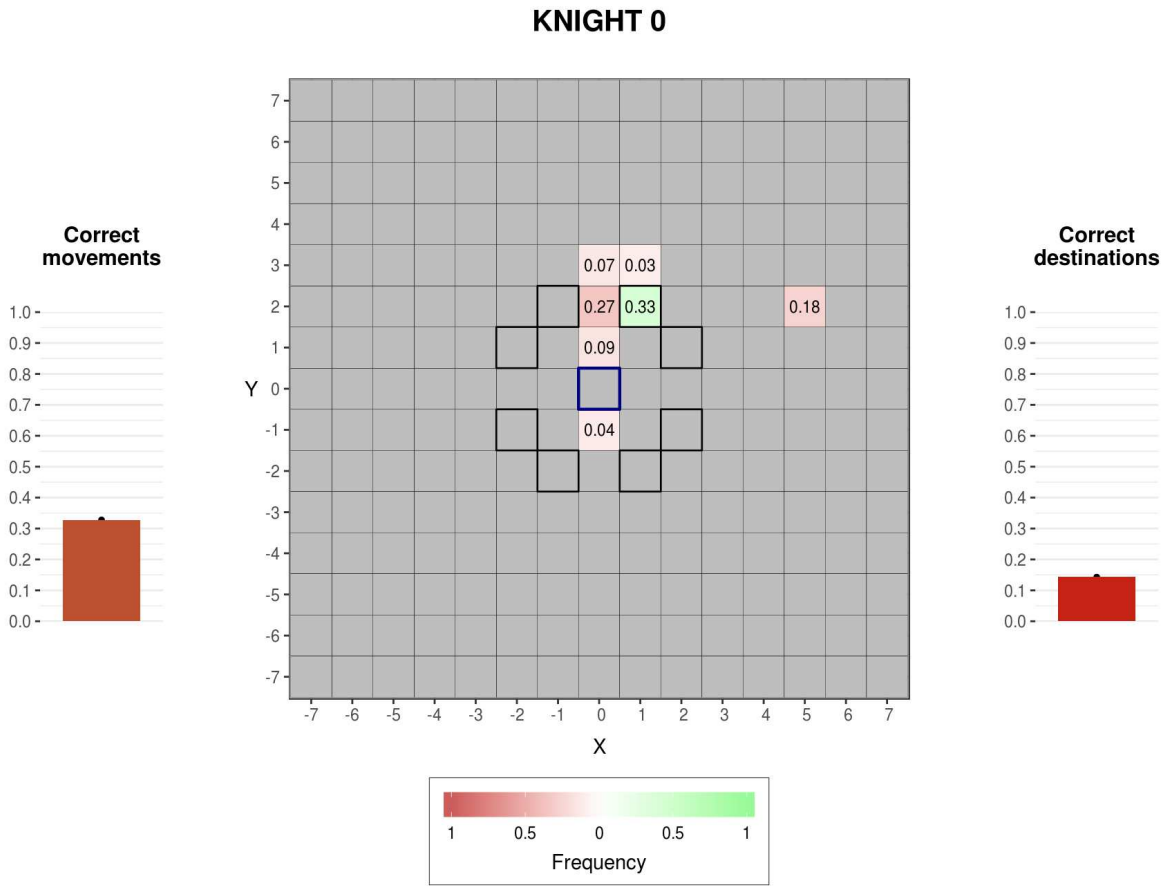


FIG. 5. This graph shows the R script schematic for Knight at snapshot 0 and base model.

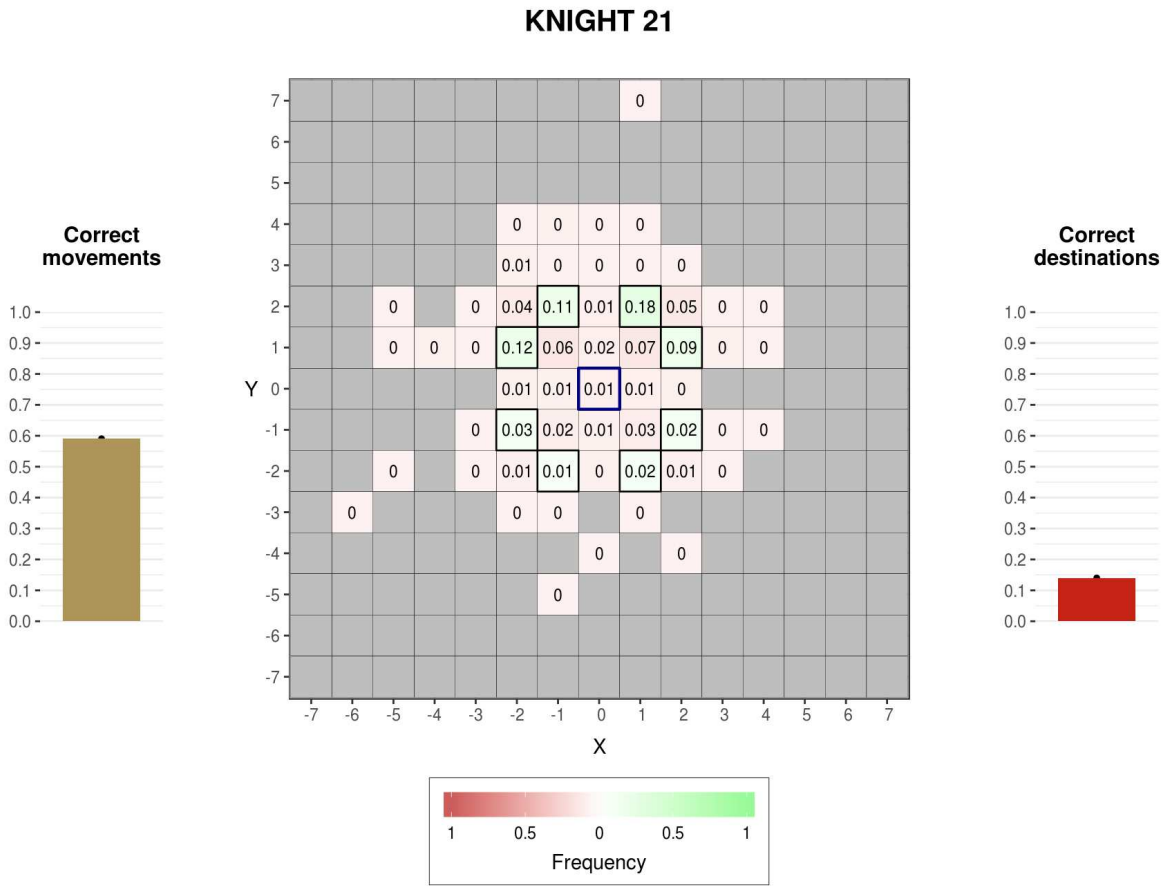


FIG. 6. This graph shows the R script schematic for Knight at snapshot 21 and base model.

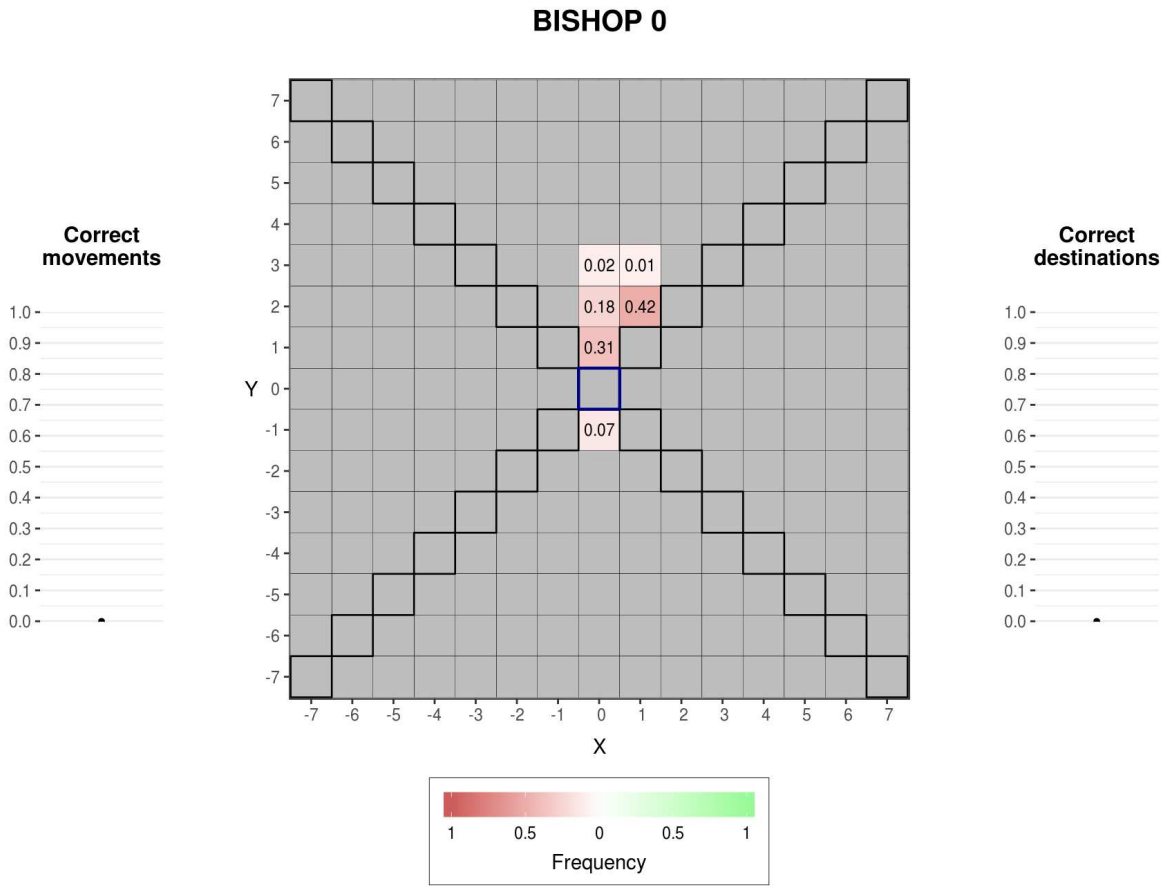


FIG. 7. This graph shows the R script schematic for Bishop at snapshot 0 and base model.

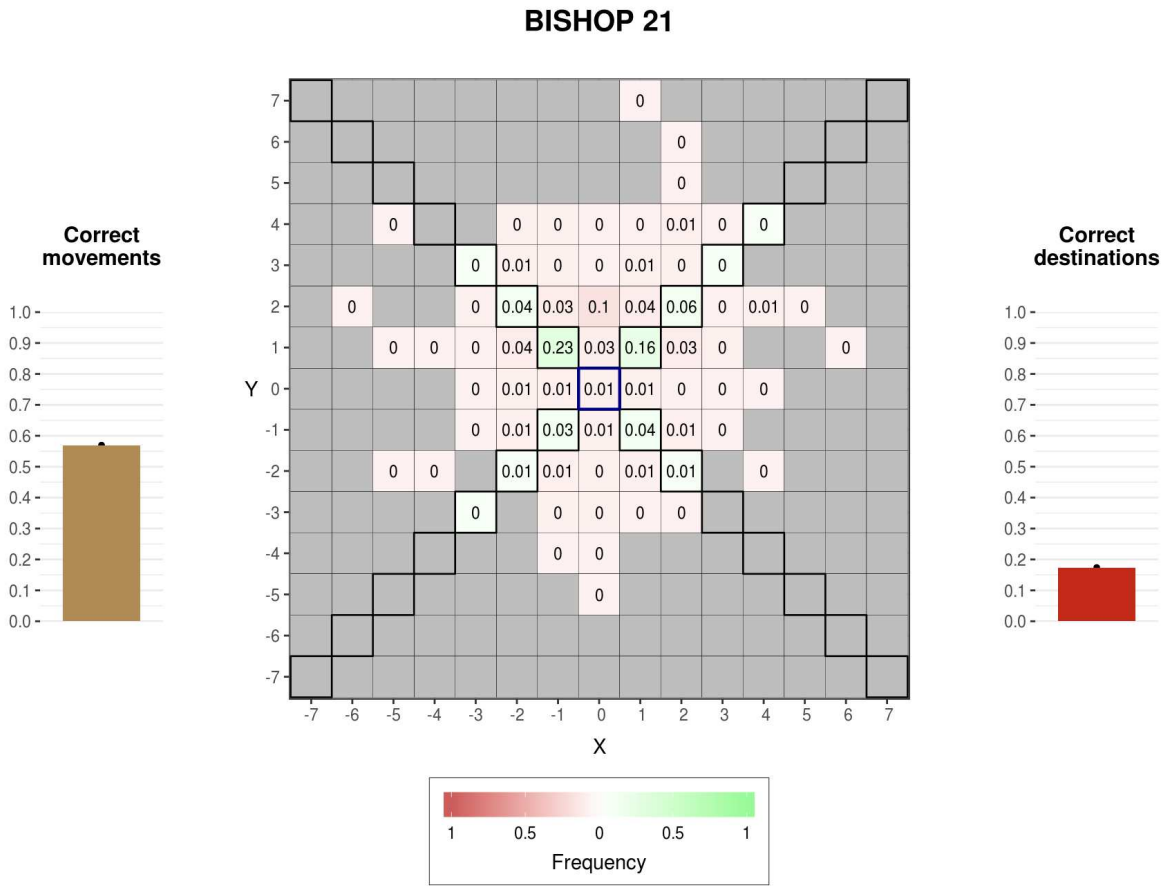


FIG. 8. This graph shows the R script schematic for Bishop at snapshot 21 and base model.

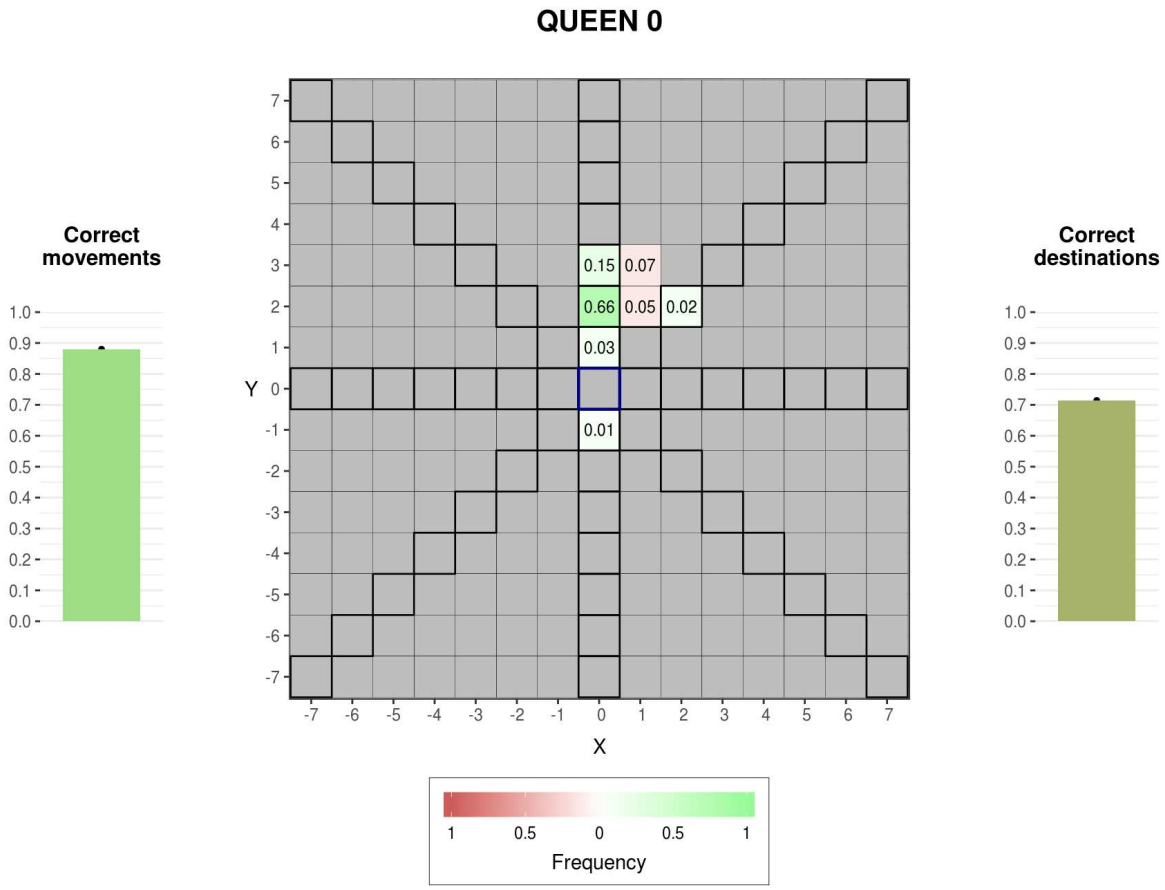


FIG. 9. This graph shows the R script schematic for Queen at snapshot 0 and base model.

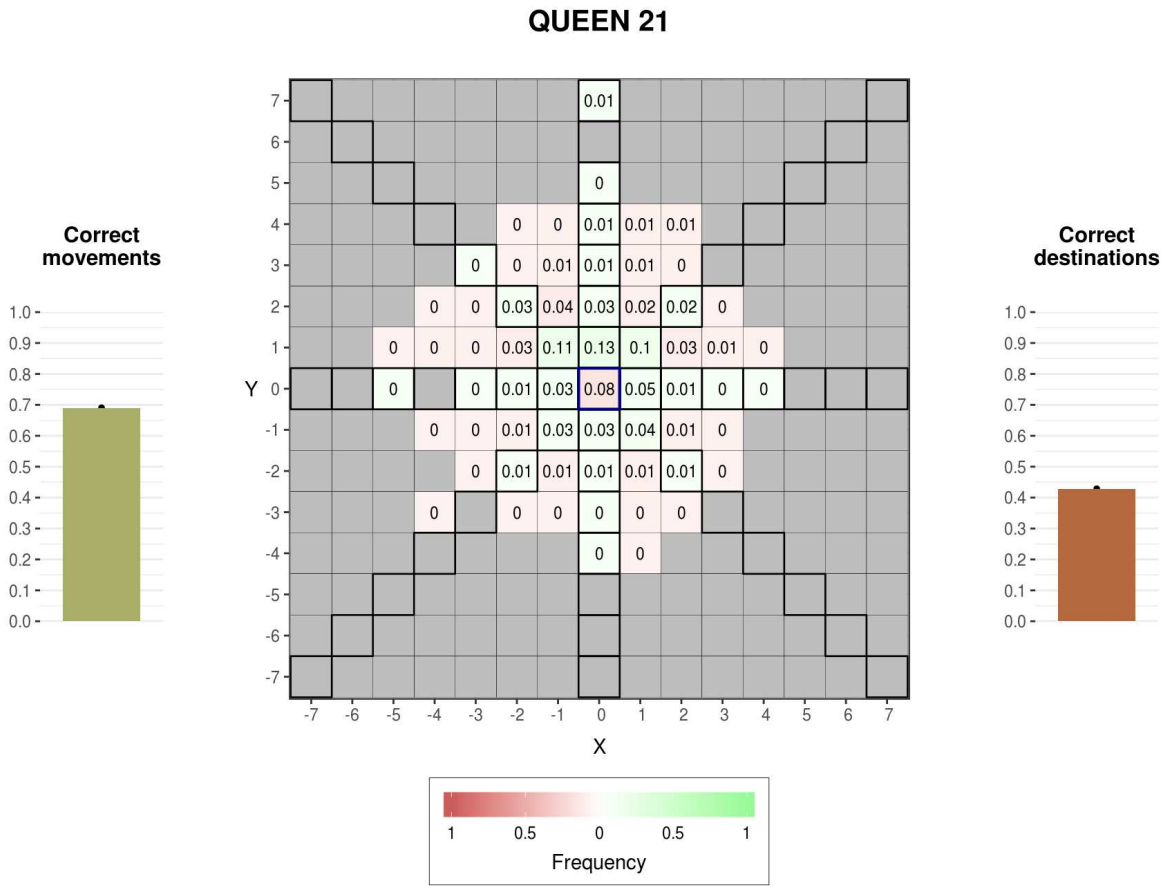


FIG. 10. This graph shows the R script schematic for Queen at snapshot 21 and base model.

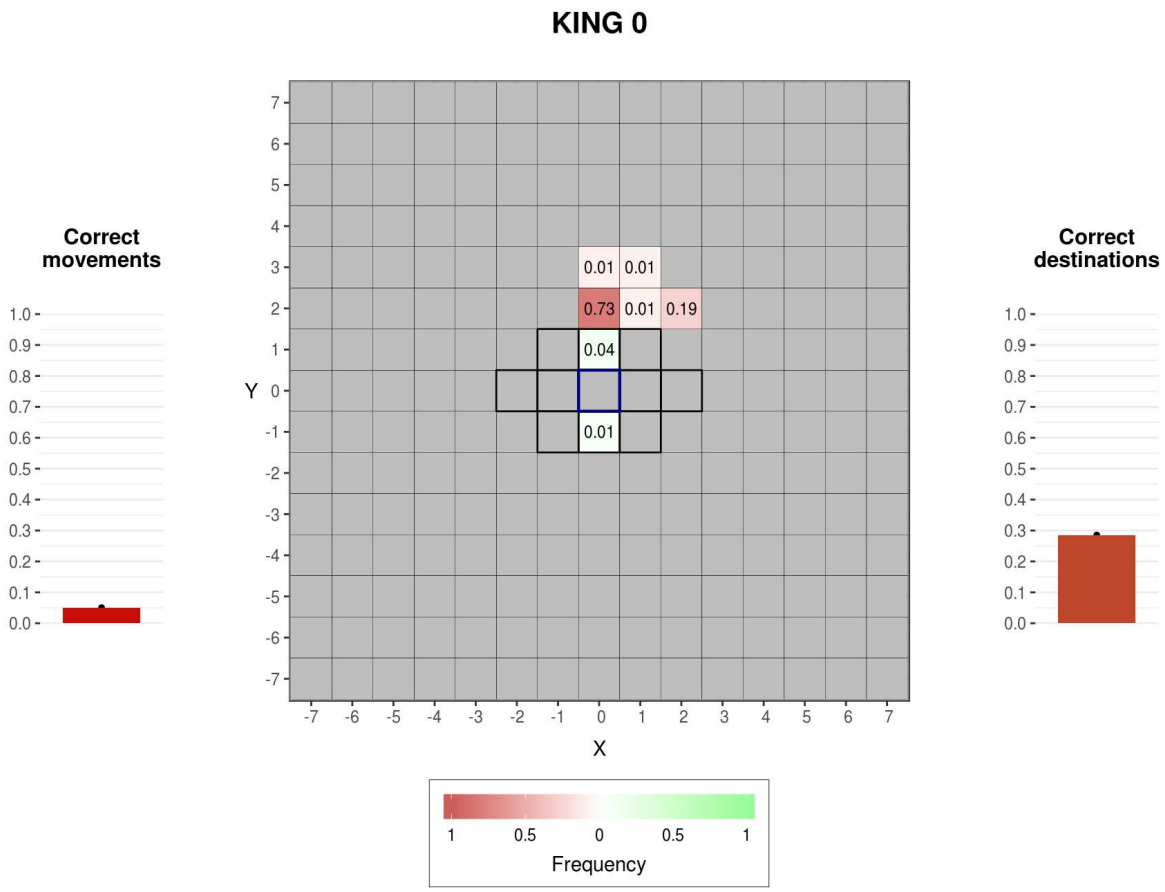


FIG. 11. This graph shows the R script schematic for King at snapshot 0 and base model.

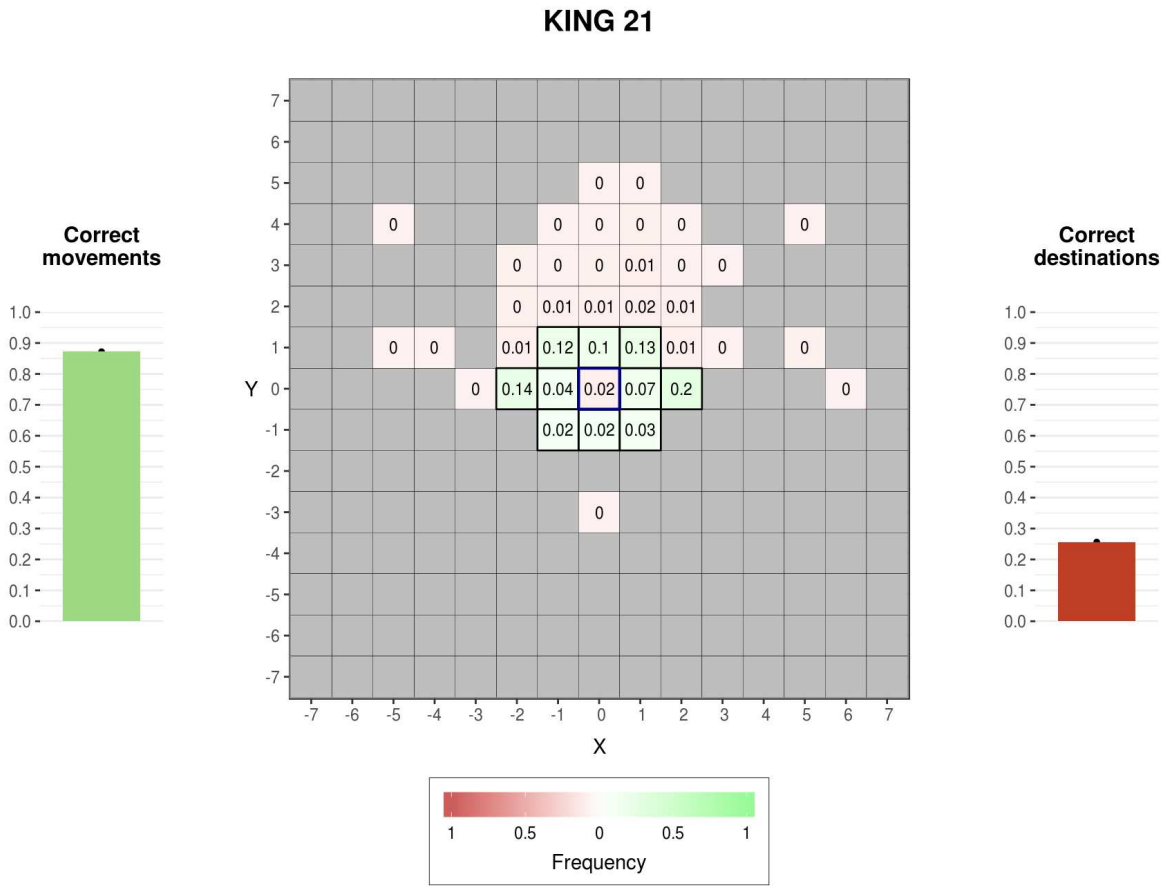


FIG. 12. This graph shows the R script schematic for King at snapshot 21 and base model.

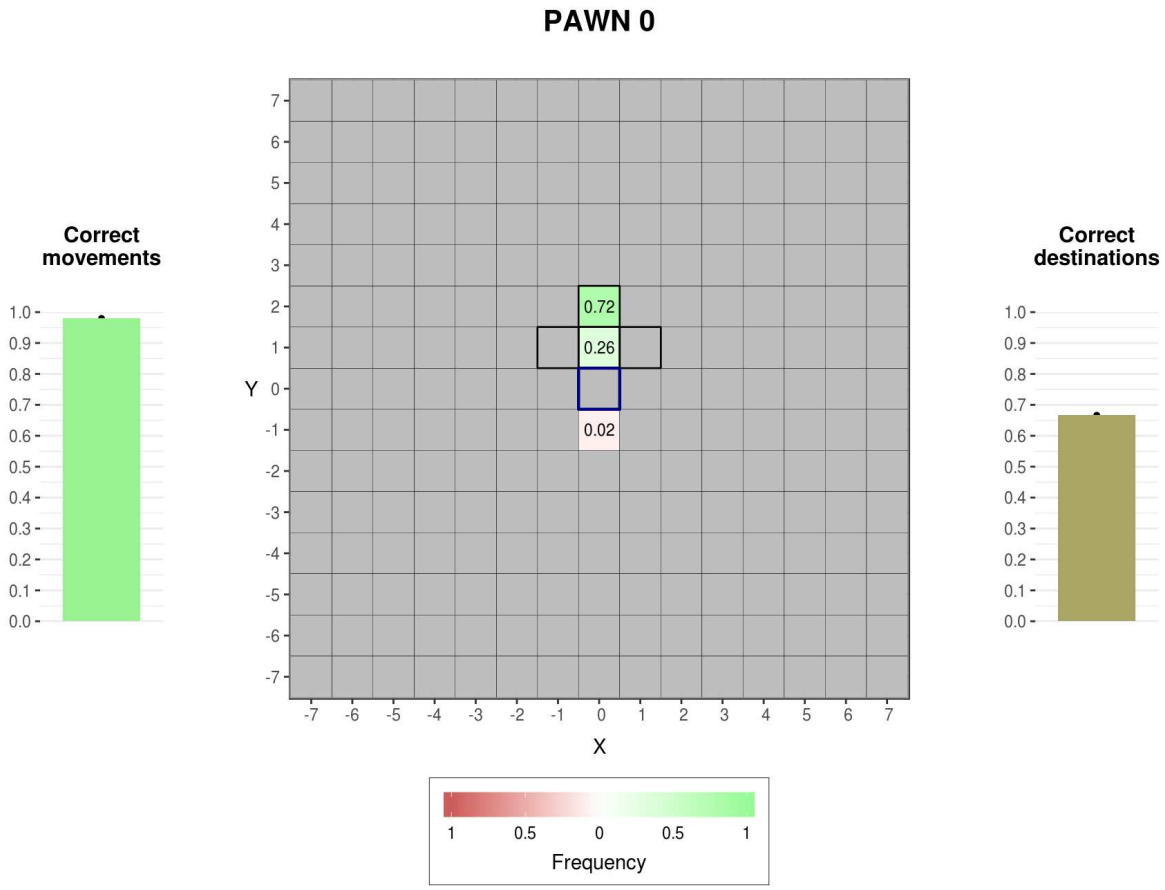


FIG. 13. This graph shows the R script schematic for Pawn at snapshot 0 and "less dense" model.

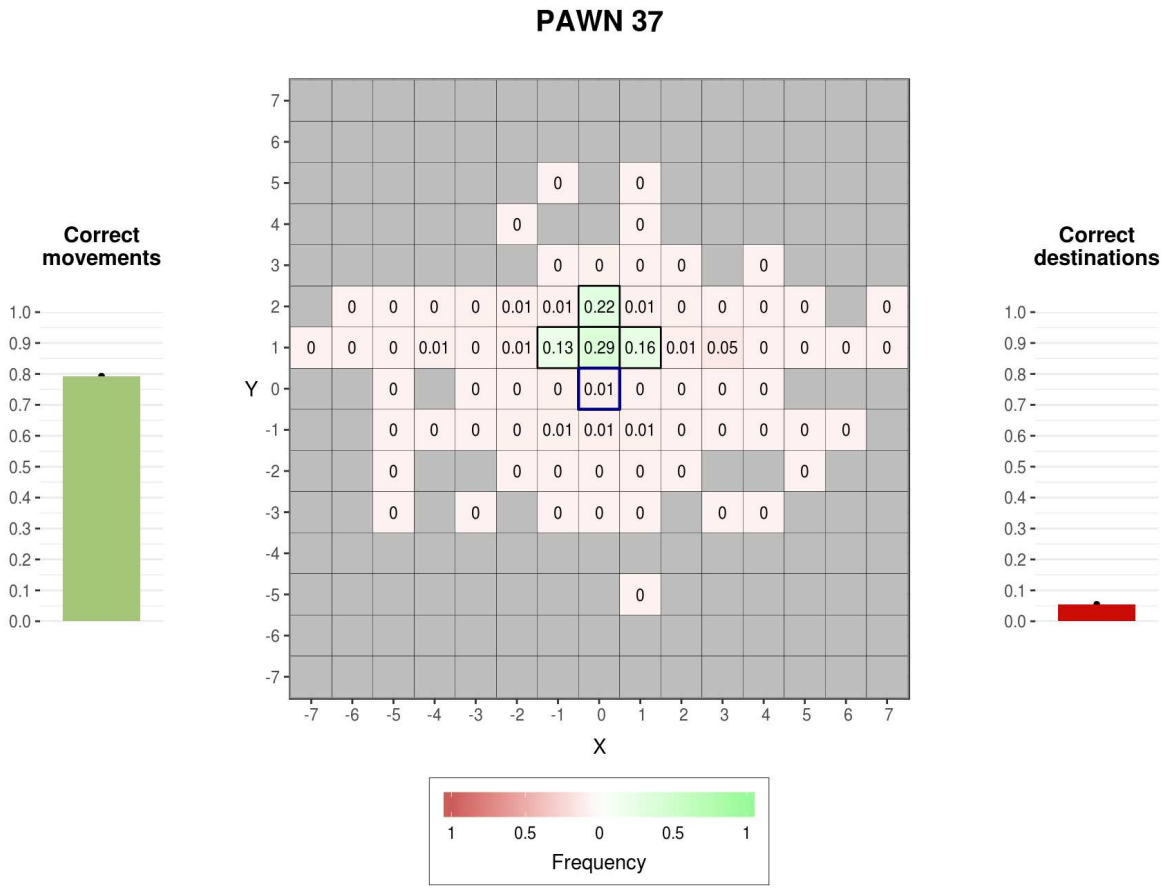


FIG. 14. This graph shows the R script schematic for Pawn at snapshot 37 and "less dense" model.

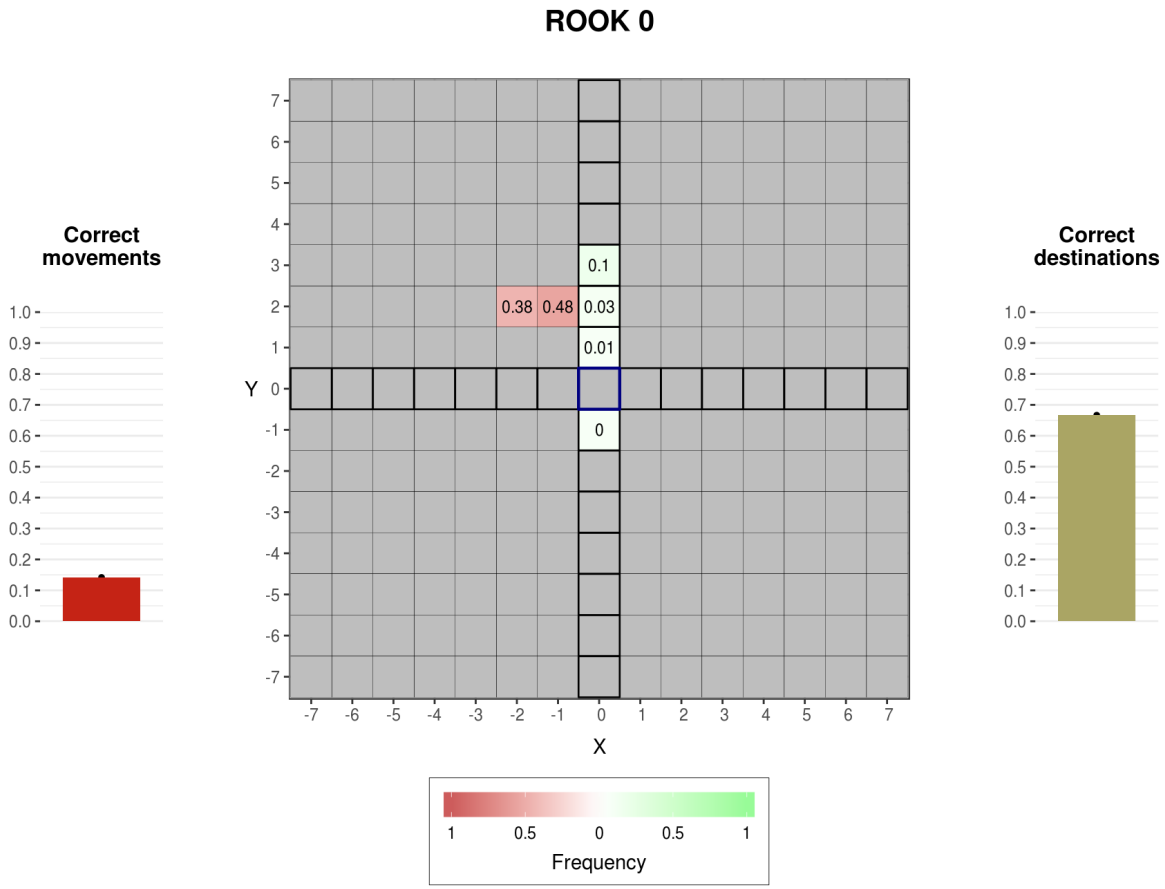


FIG. 15. This graph shows the R script schematic for Rook at snapshot 0 and "less dense" model.

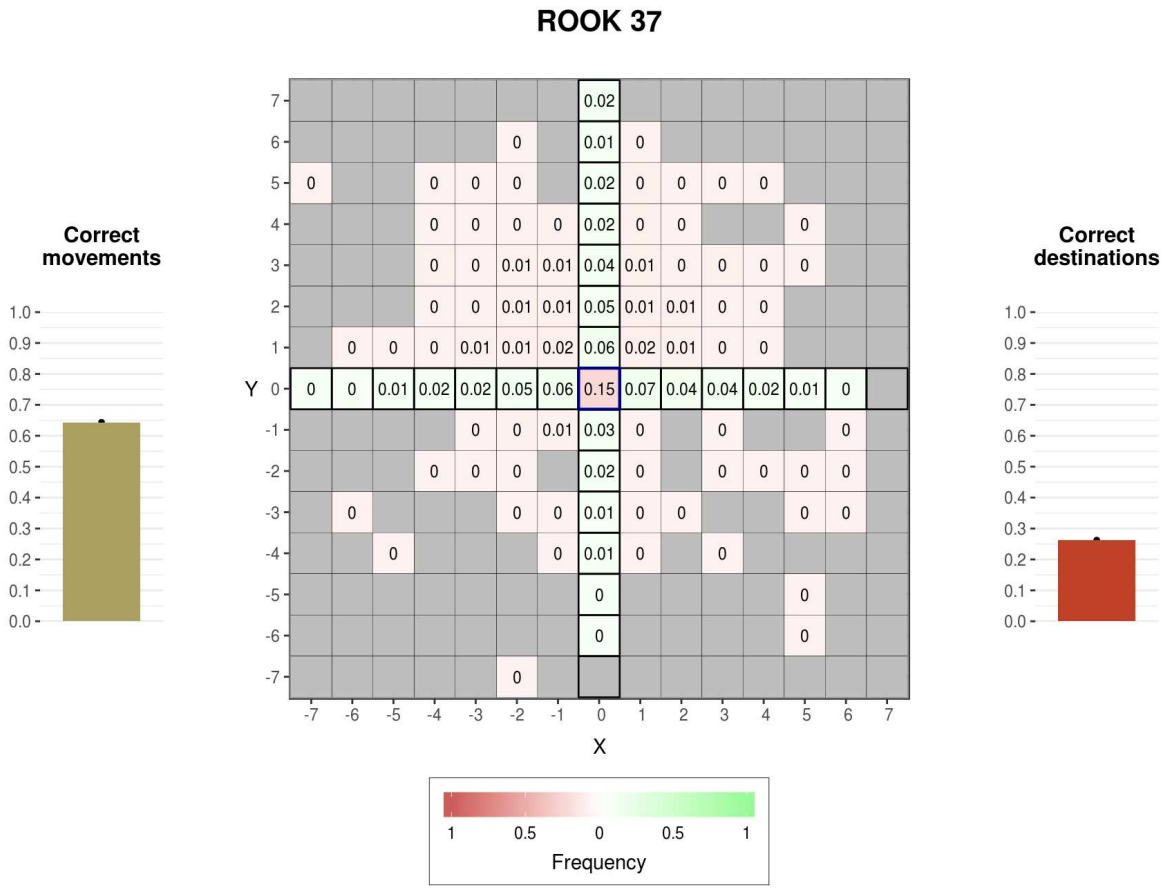


FIG. 16. This graph shows the R script schematic for Rook at snapshot 37 and "less dense" model.

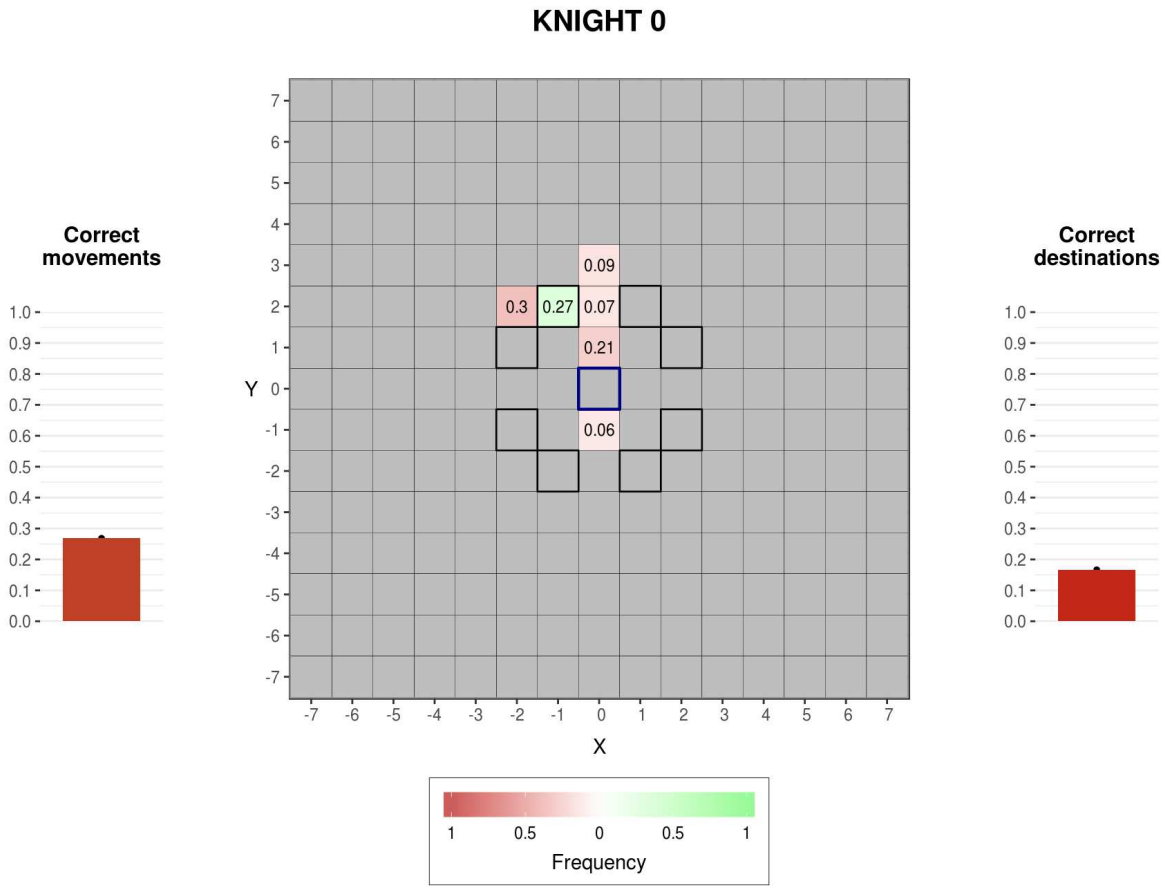


FIG. 17. This graph shows the R script schematic for Knight at snapshot 0 and "less dense" model.

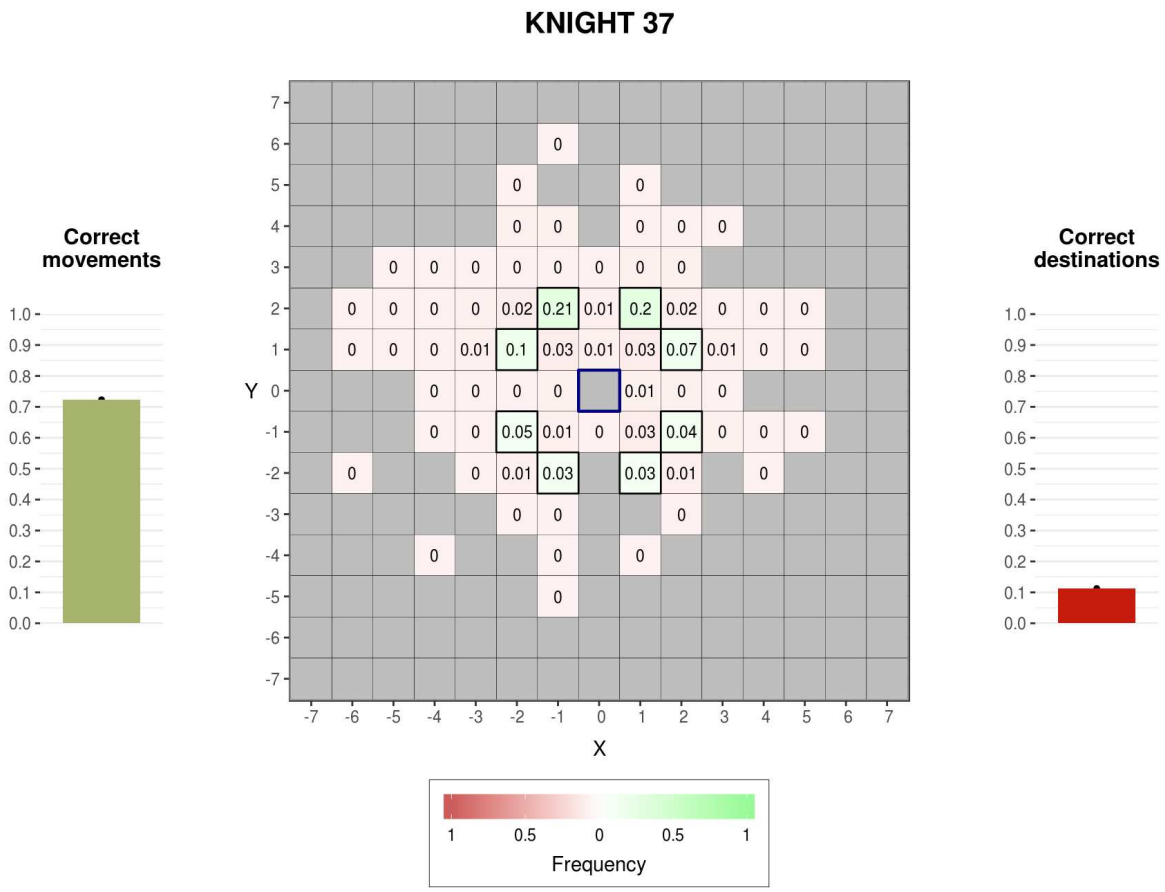


FIG. 18. This graph shows the R script schematic for Knight at snapshot 37 and "less dense" model.

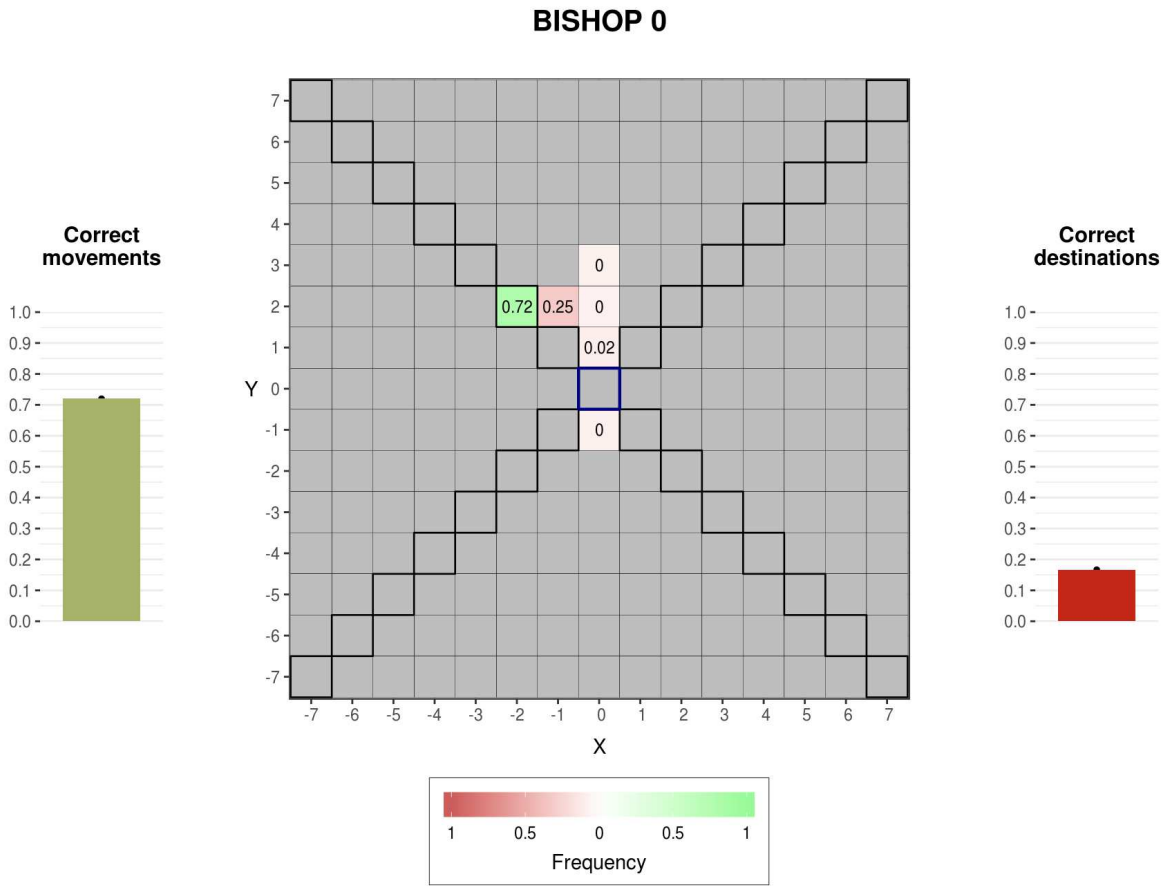


FIG. 19. This graph shows the R script schematic for Bishop at snapshot 0 and "less dense" model.

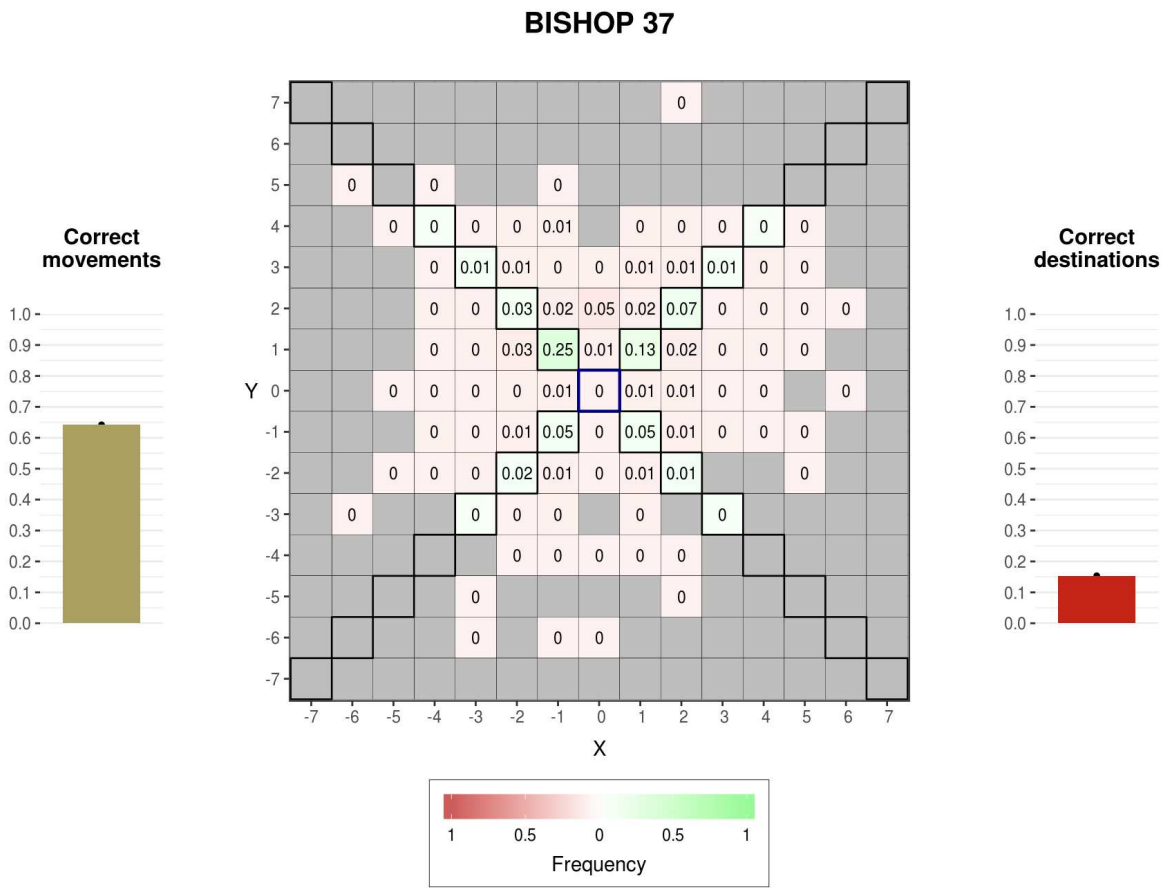


FIG. 20. This graph shows the R script schematic for Bishop at snapshot 37 and "less dense" model.

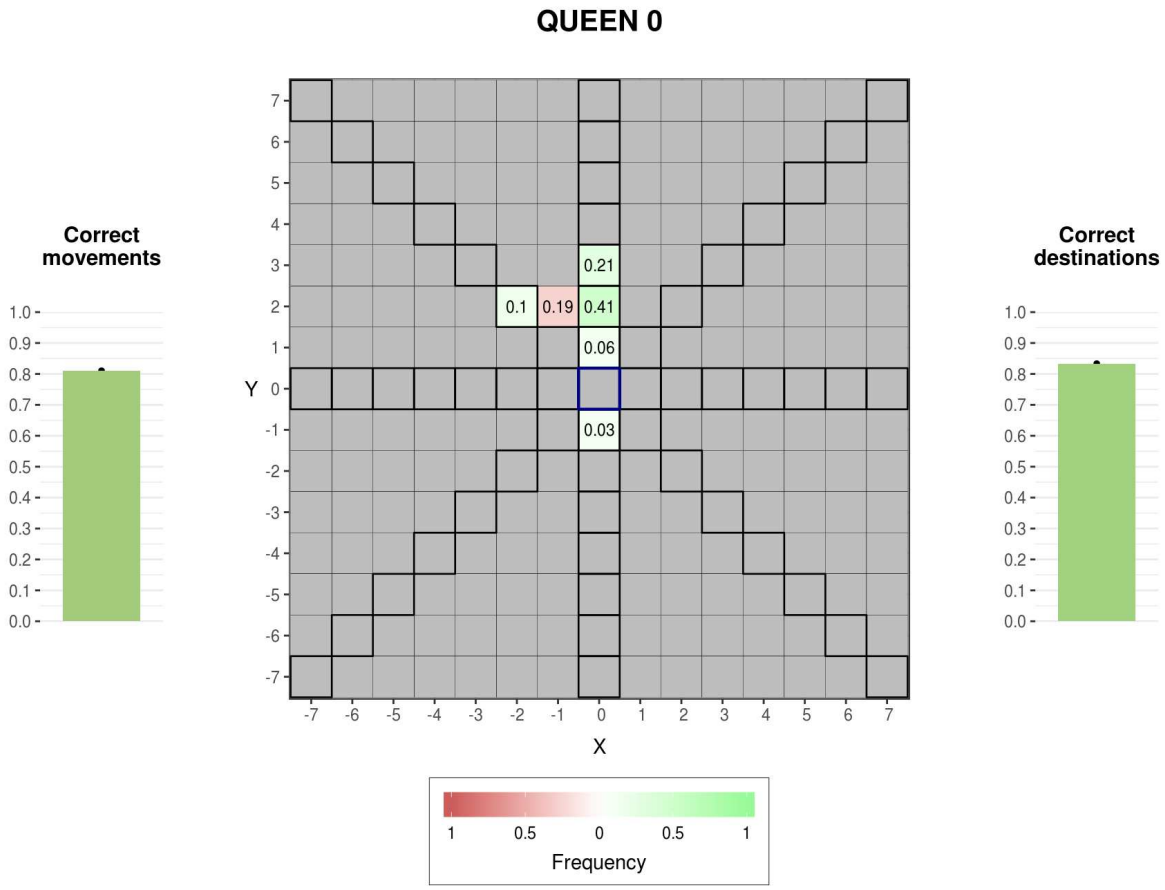


FIG. 21. This graph shows the R script schematic for Queen at snapshot 0 and "less dense" model.

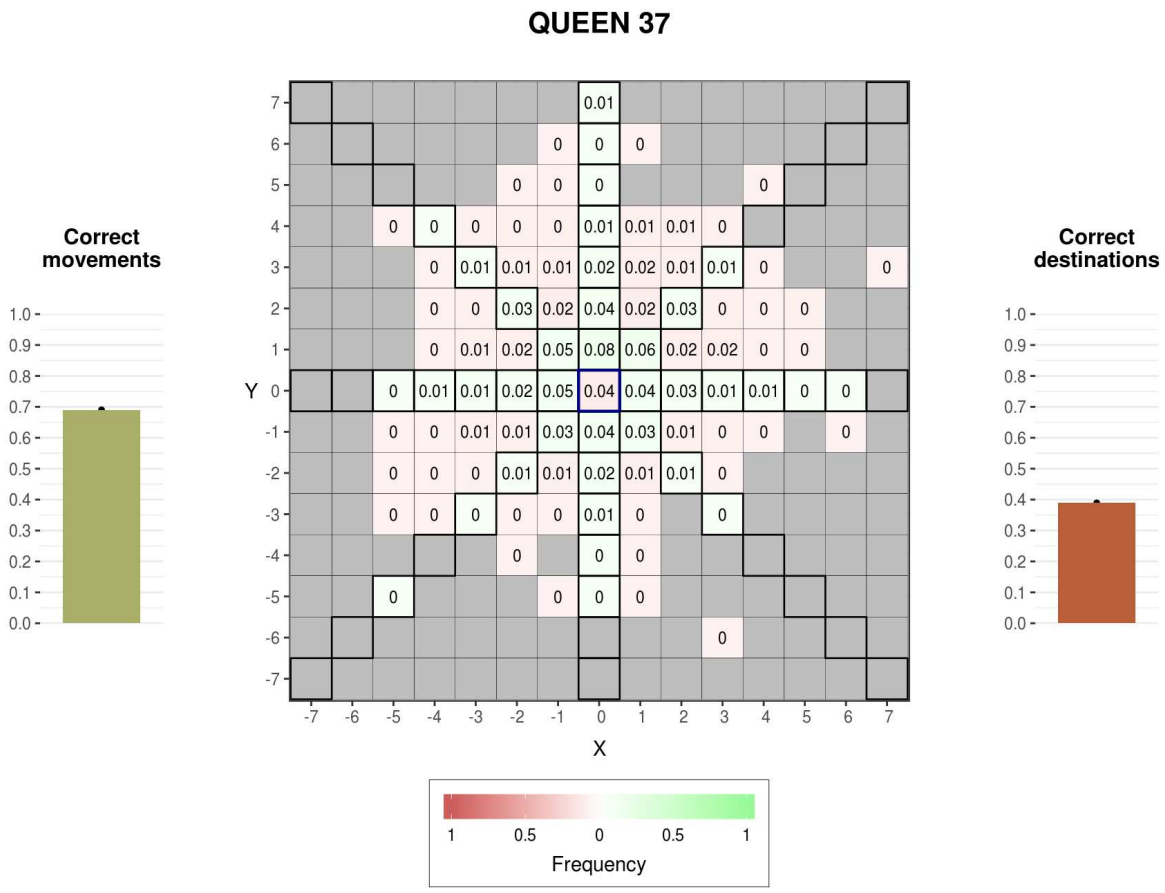


FIG. 22. This graph shows the R script schematic for Queen at snapshot 37 and "less dense" model.

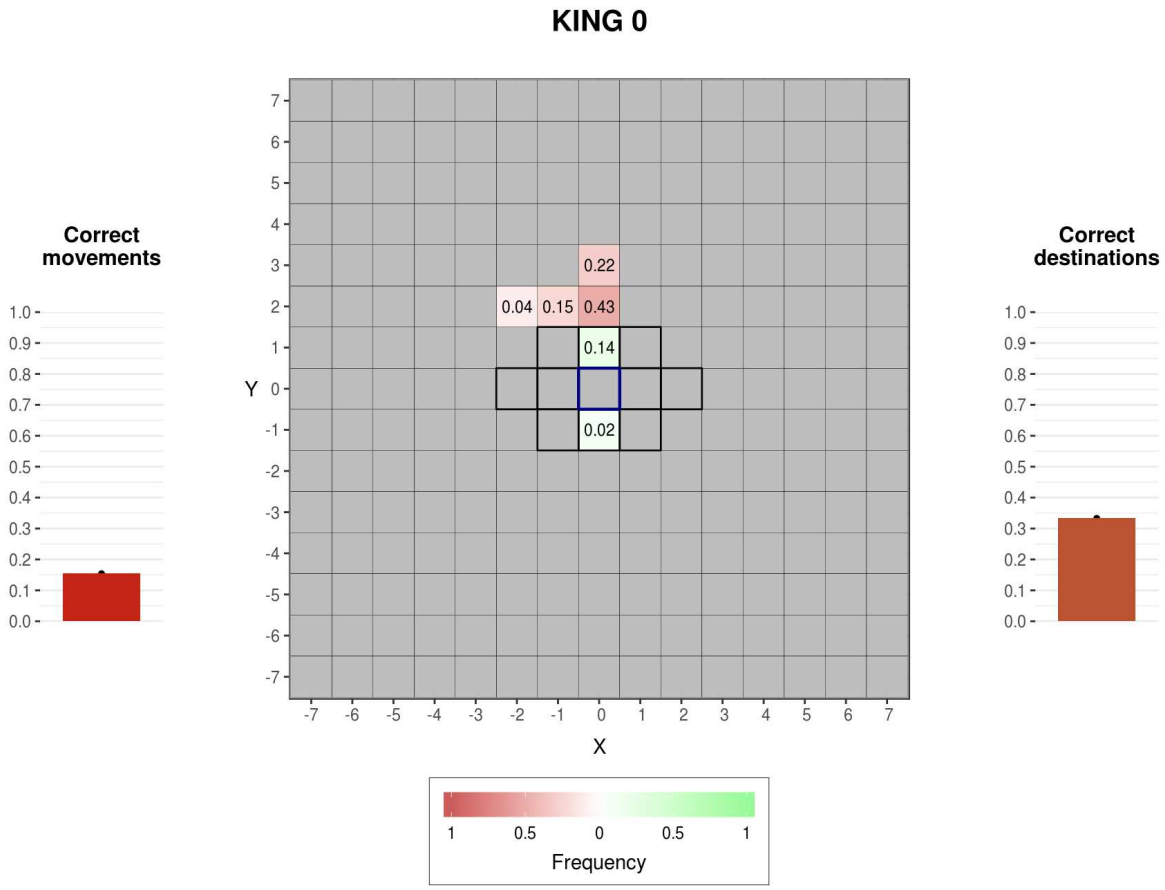


FIG. 23. This graph shows the R script schematic for King at snapshot 0 and "less dense" model.

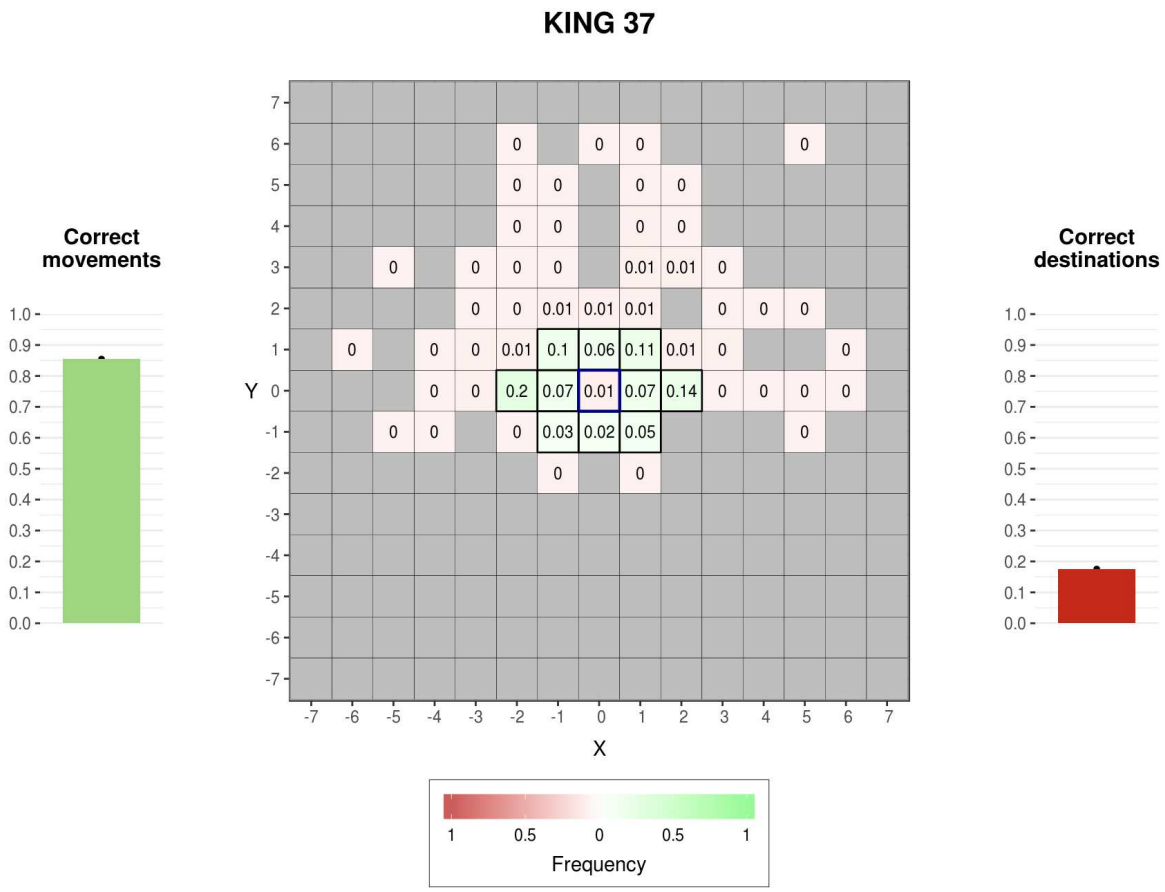


FIG. 24. This graph shows the R script schematic for King at snapshot 37 and "less dense" model.

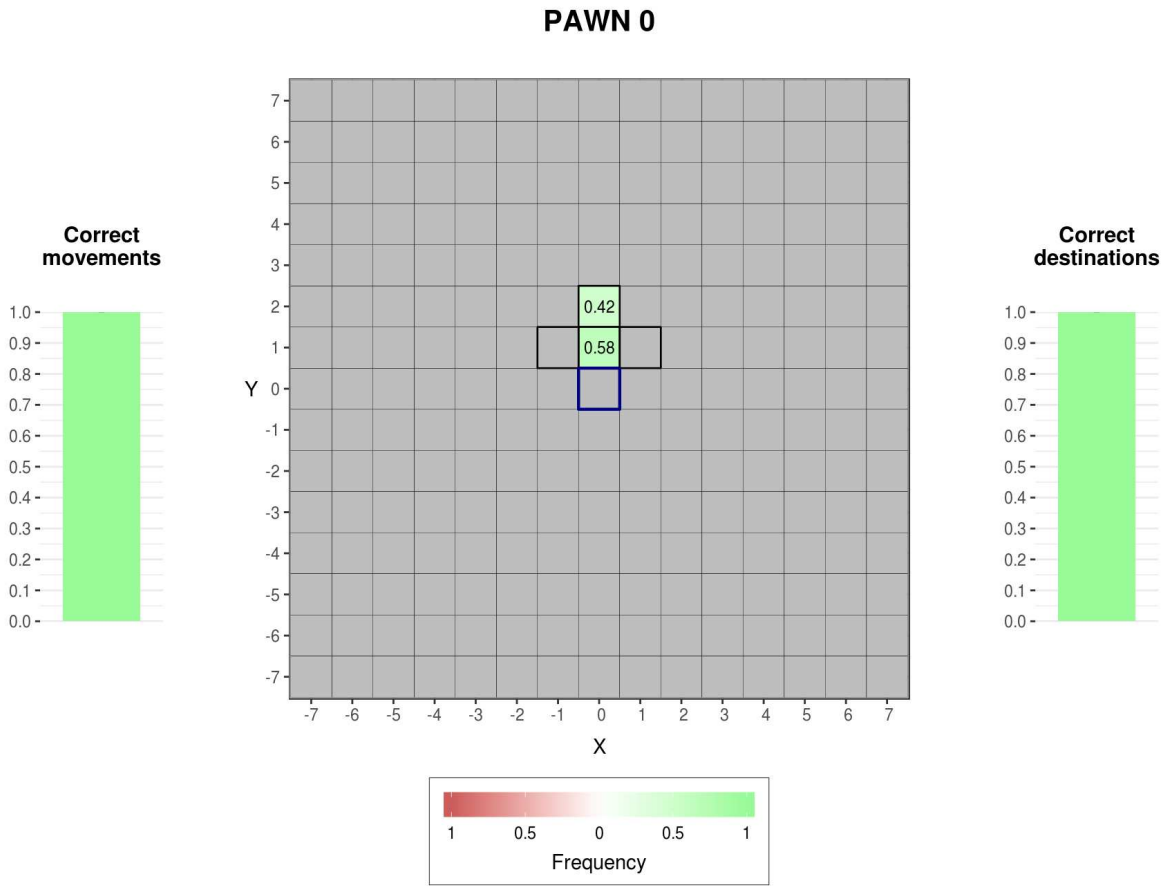


FIG. 25. This graph shows the R script schematic for Pawn at snapshot 0 and "more dense" model.

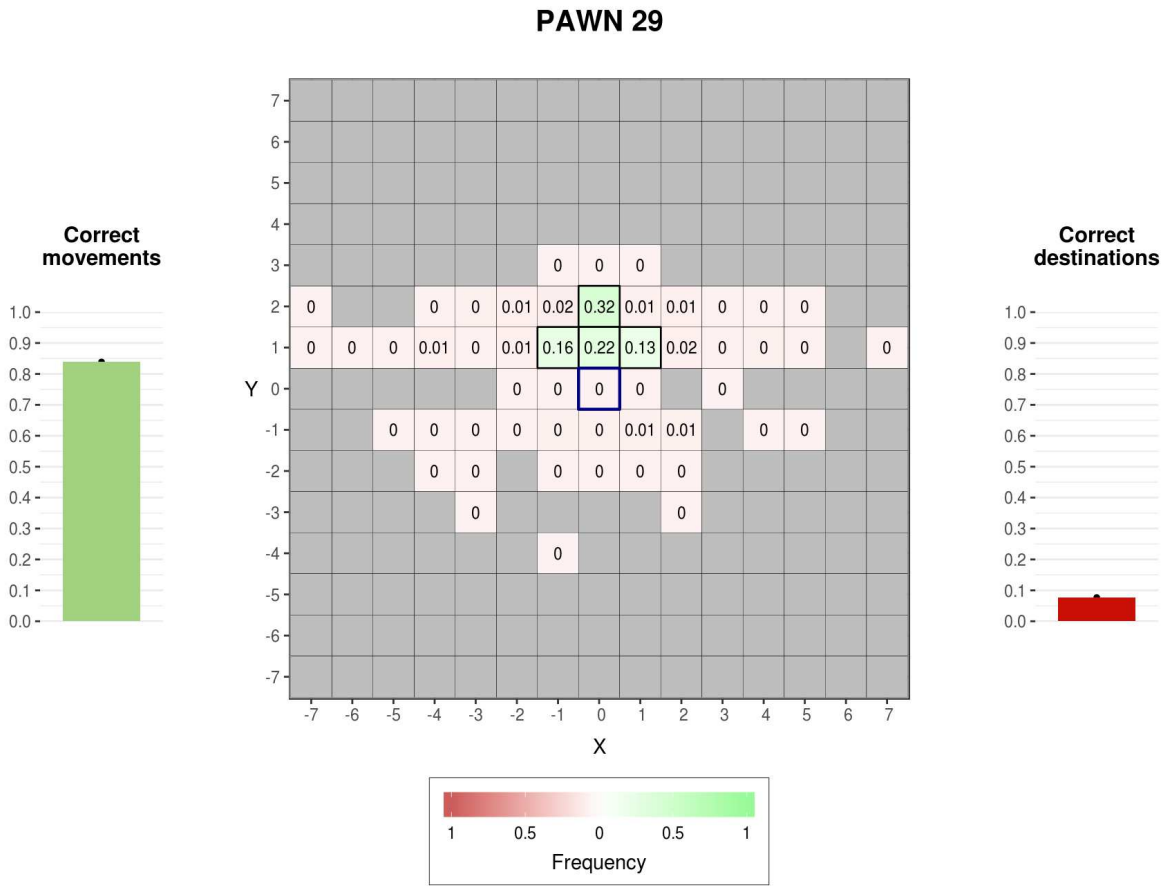


FIG. 26. This graph shows the R script schematic for Pawn at snapshot 29 and "more dense" model.

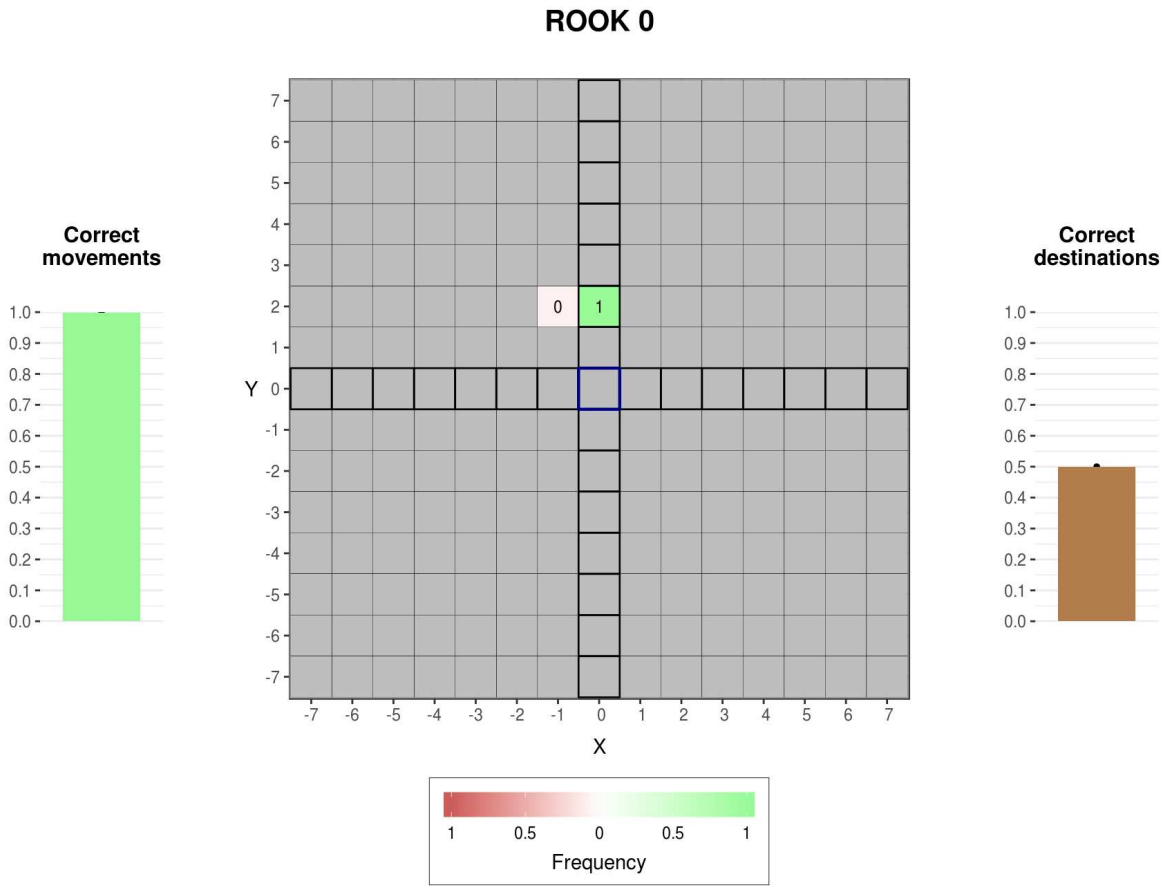


FIG. 27. This graph shows the R script schematic for Rook at snapshot 0 and "more dense" model.

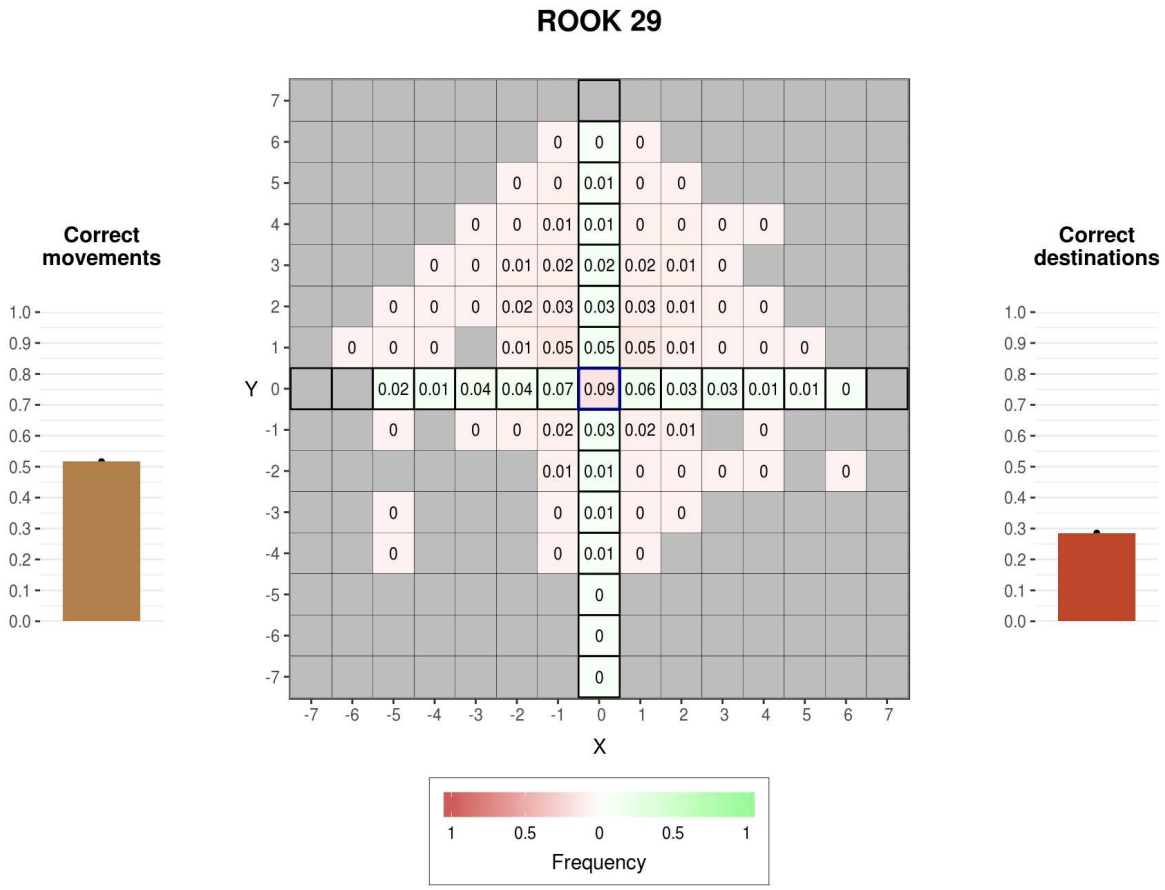


FIG. 28. This graph shows the R script schematic for Rook at snapshot 29 and "more dense" model.

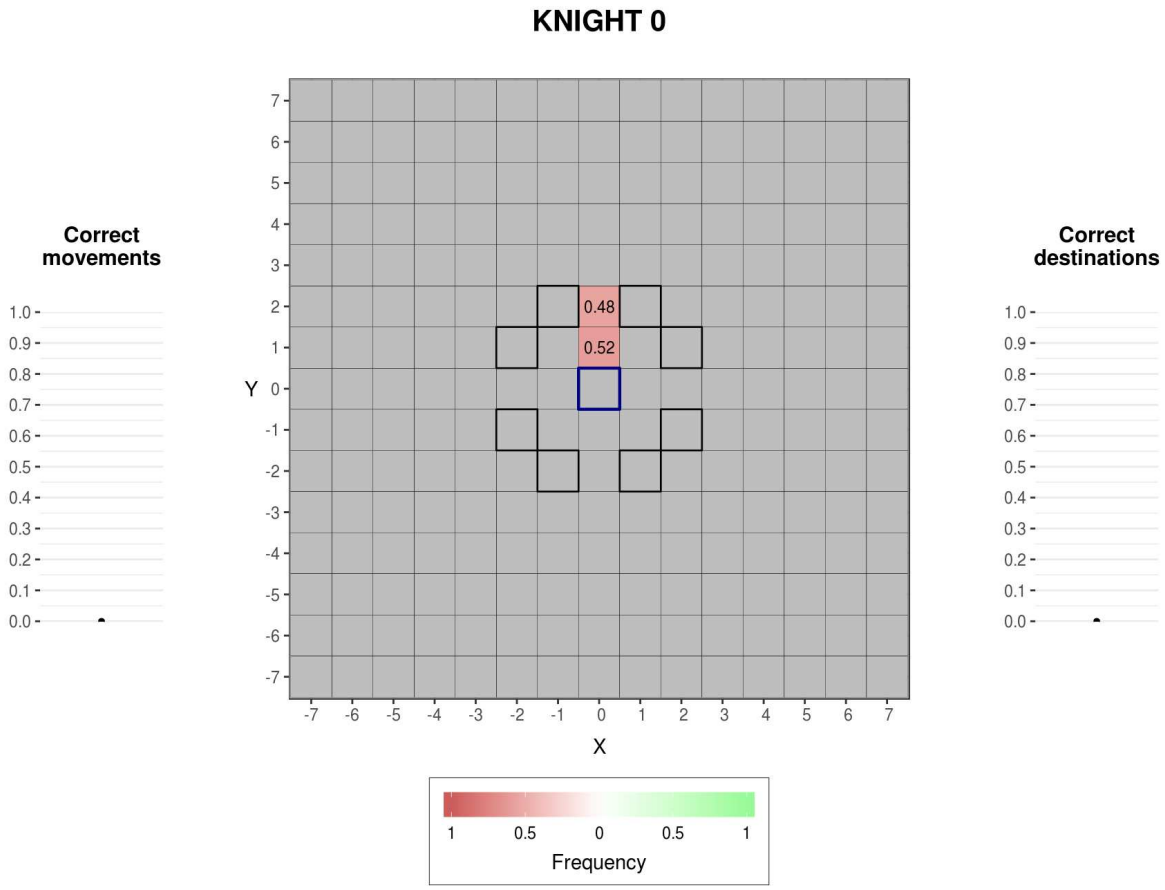


FIG. 29. This graph shows the R script schematic for Knight at snapshot 0 and "more dense" model.

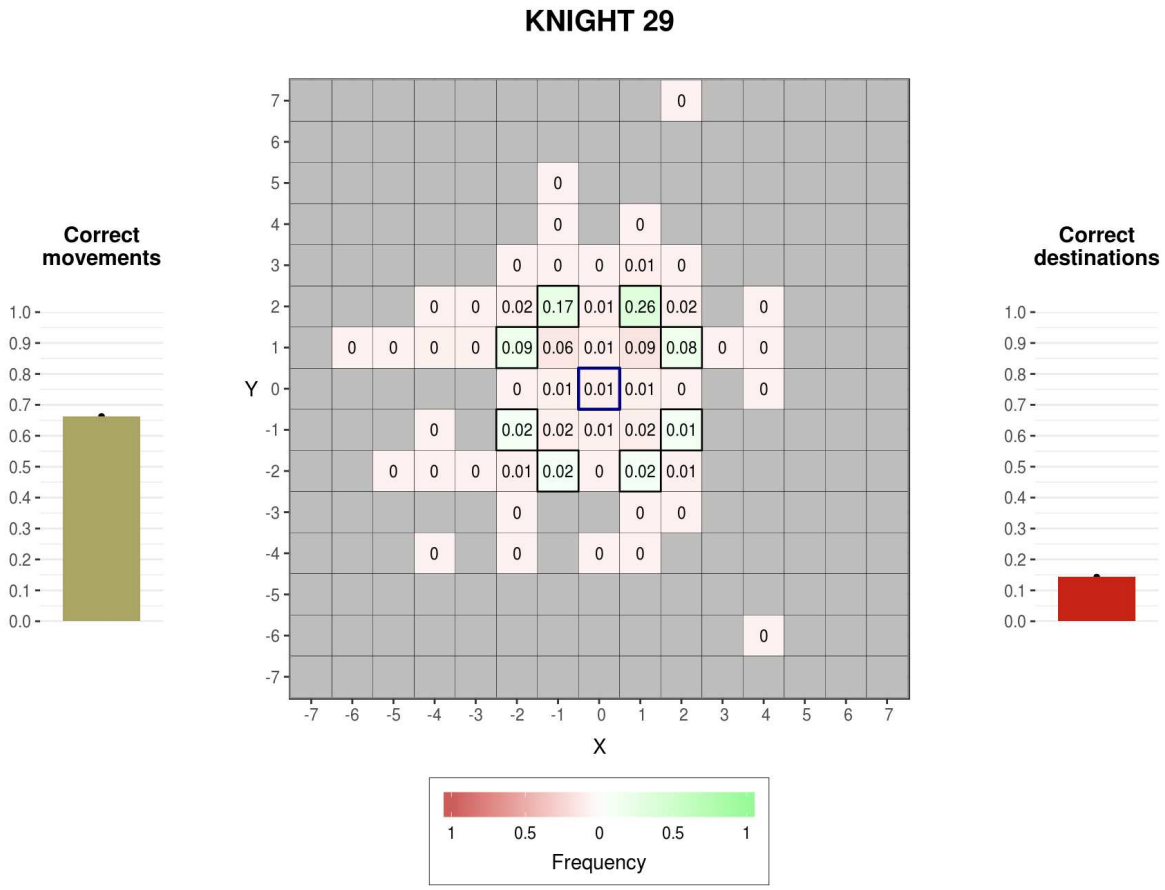


FIG. 30. This graph shows the R script schematic for Knight at snapshot 29 and "more dense" model.

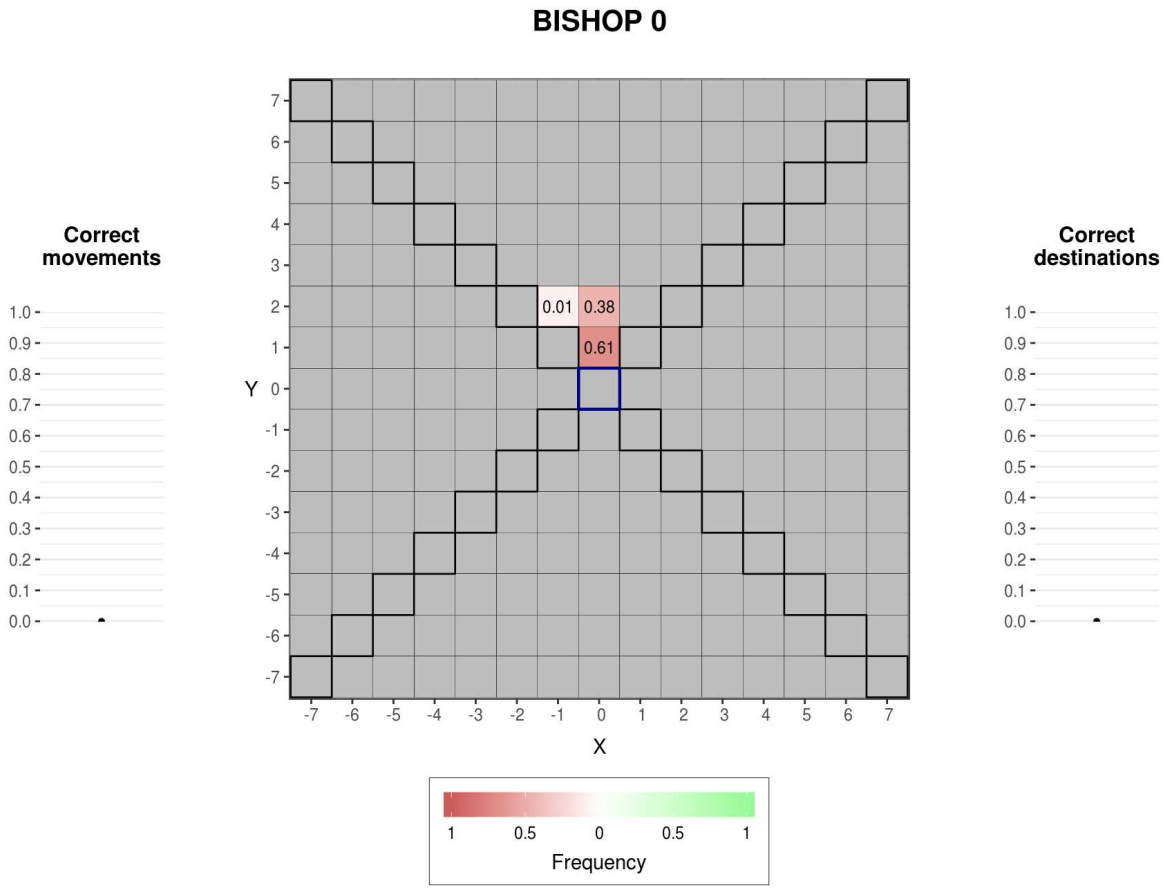


FIG. 31. This graph shows the R script schematic for Bishop at snapshot 0 and "more dense" model.

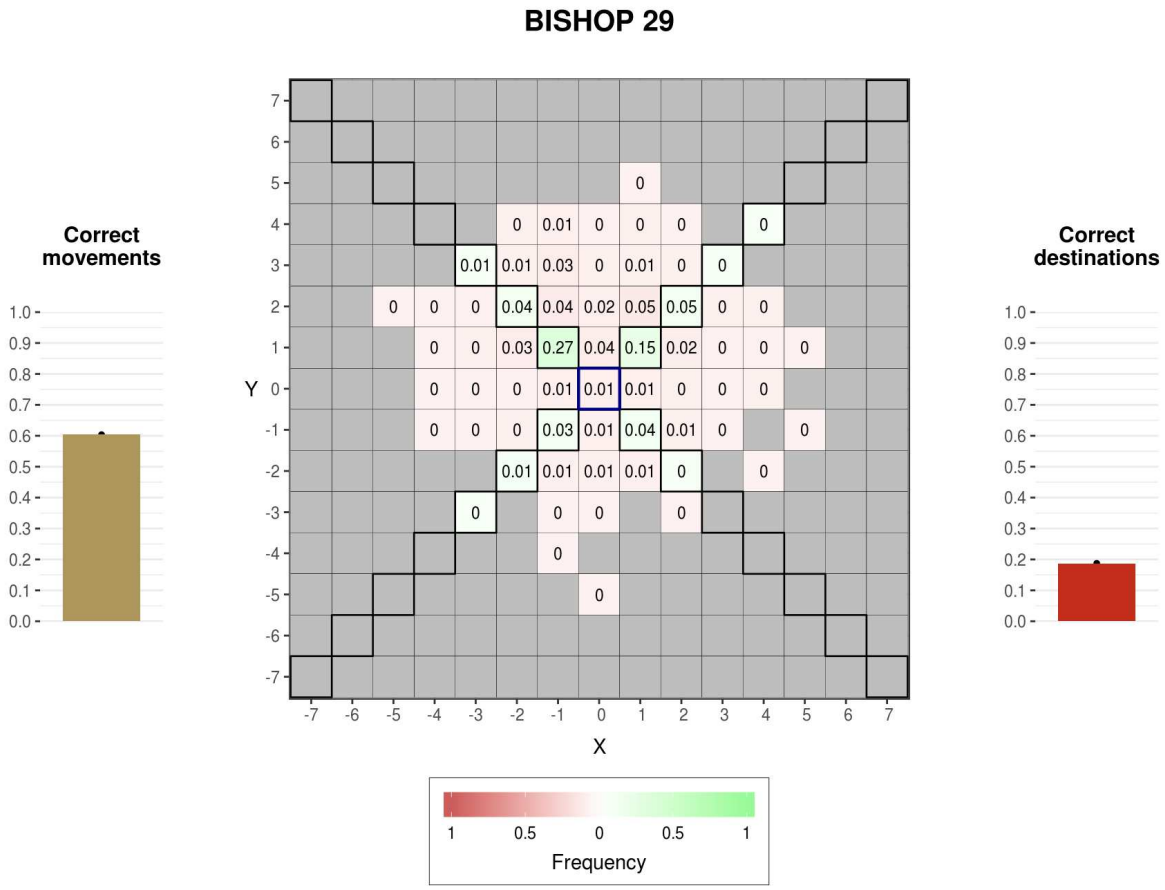


FIG. 32. This graph shows the R script schematic for Bishop at snapshot 29 and "more dense" model.

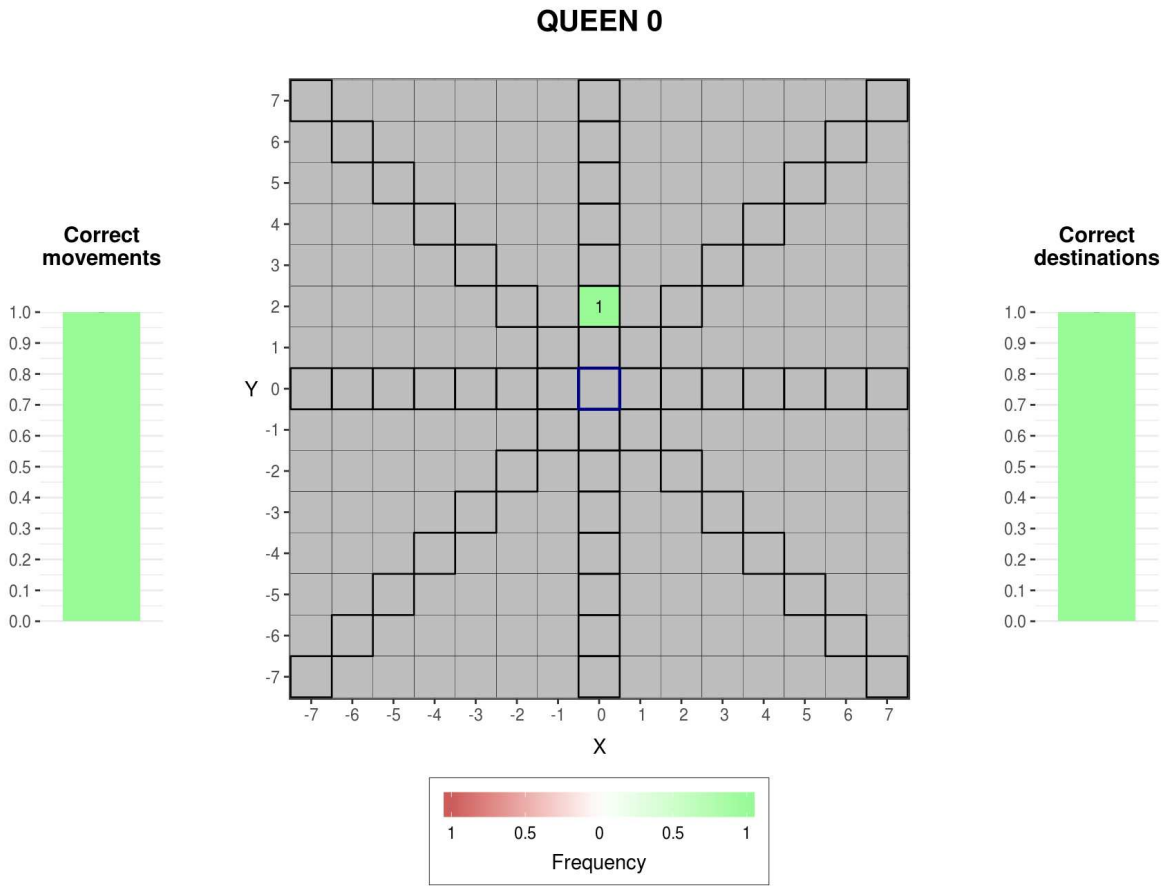


FIG. 33. This graph shows the R script schematic for Queen at snapshot 0 and "more dense" model.

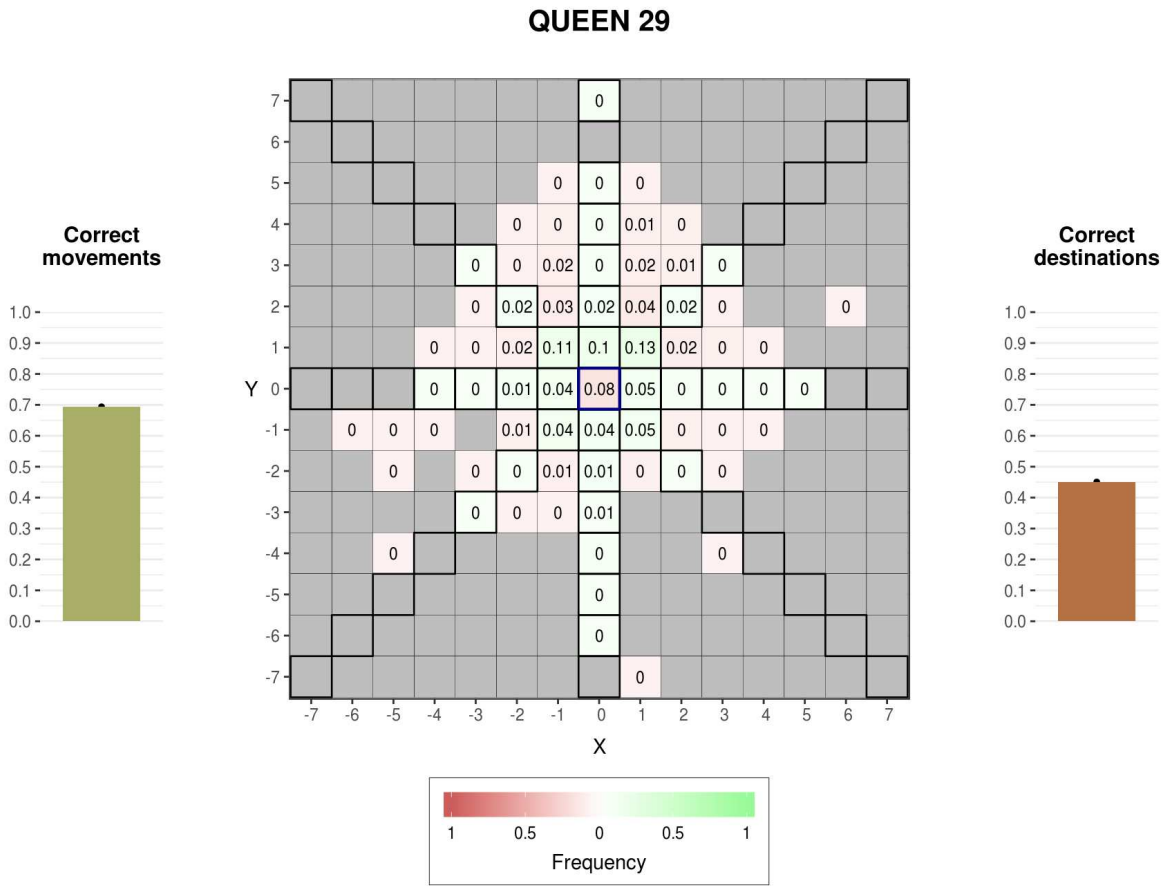


FIG. 34. This graph shows the R script schematic for Queen at snapshot 29 and "more dense" model.

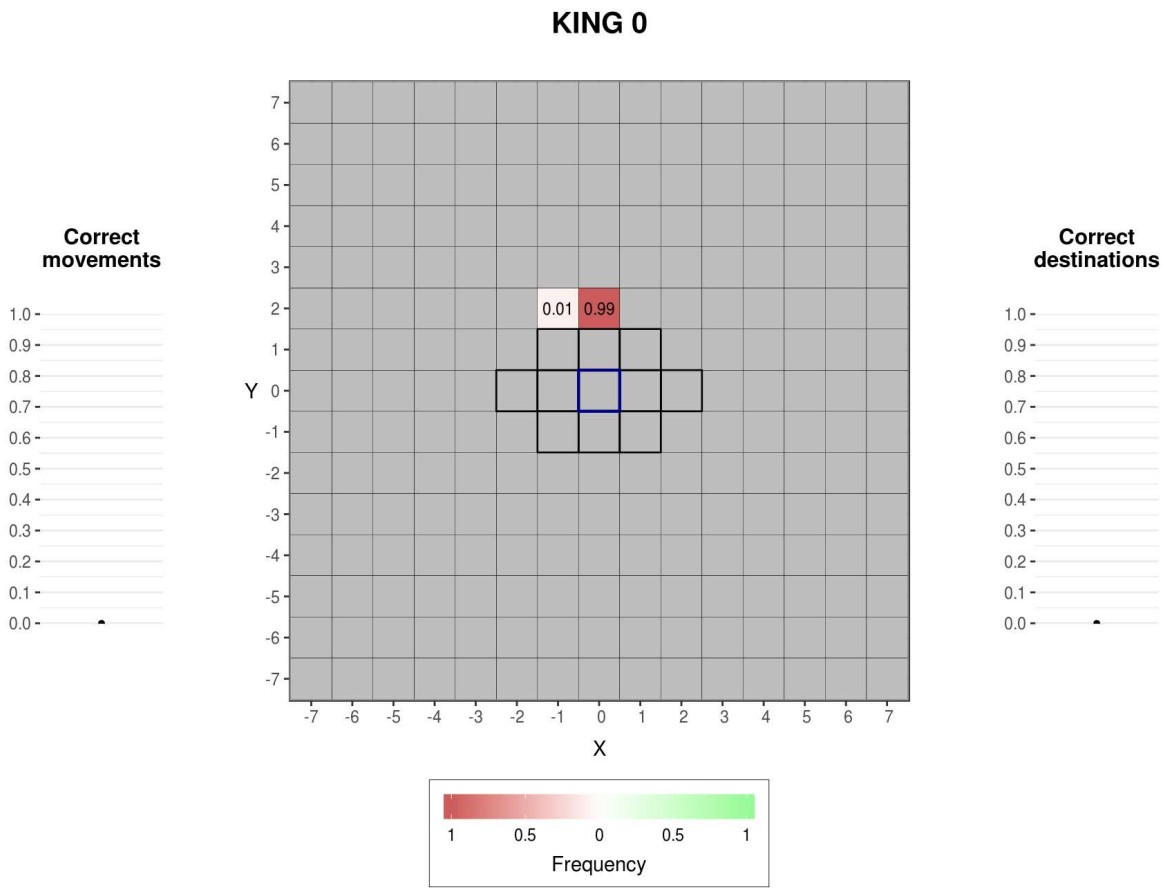


FIG. 35. This graph shows the R script schematic for King at snapshot 0 and "more dense" model.

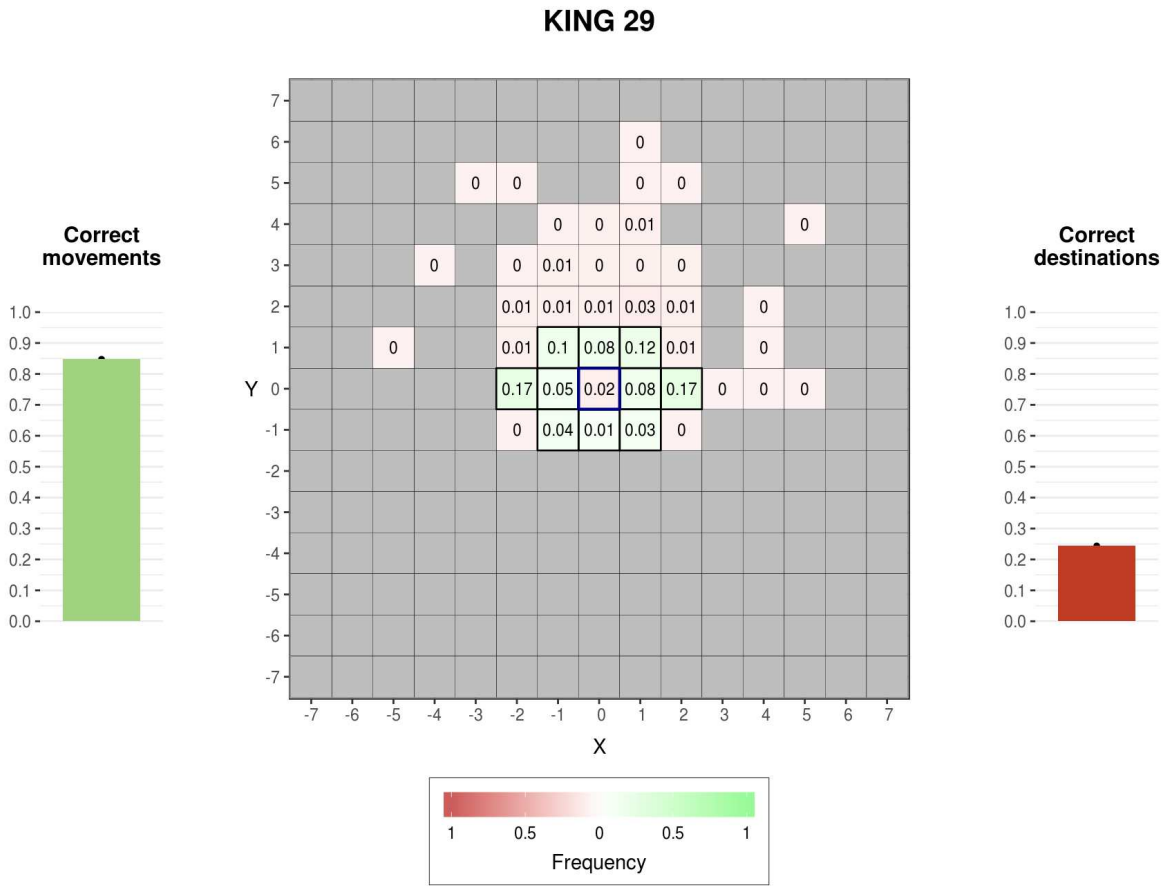


FIG. 36. This graph shows the R script schematic for King at snapshot 29 and "more dense" model.

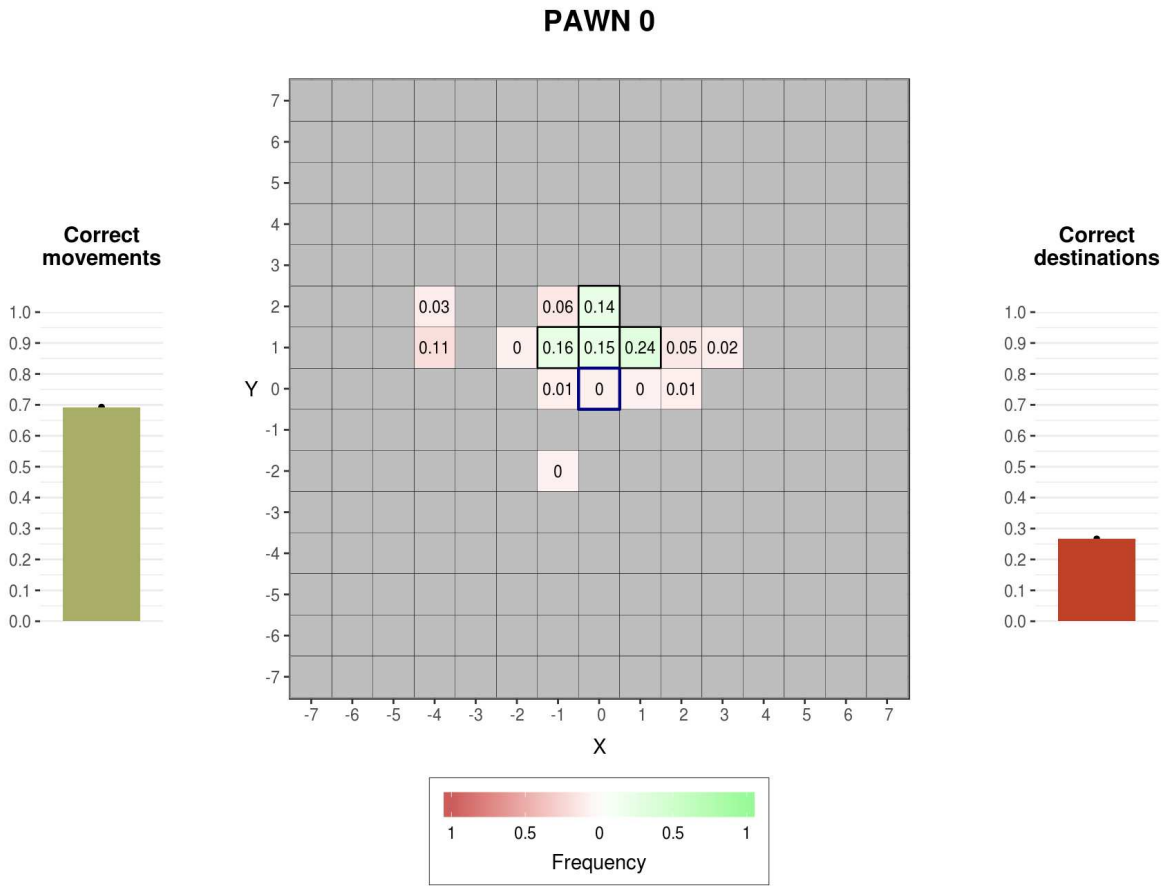


FIG. 37. This graph shows the R script schematic for Pawn at snapshot 0 and "less convolutional layers" model.

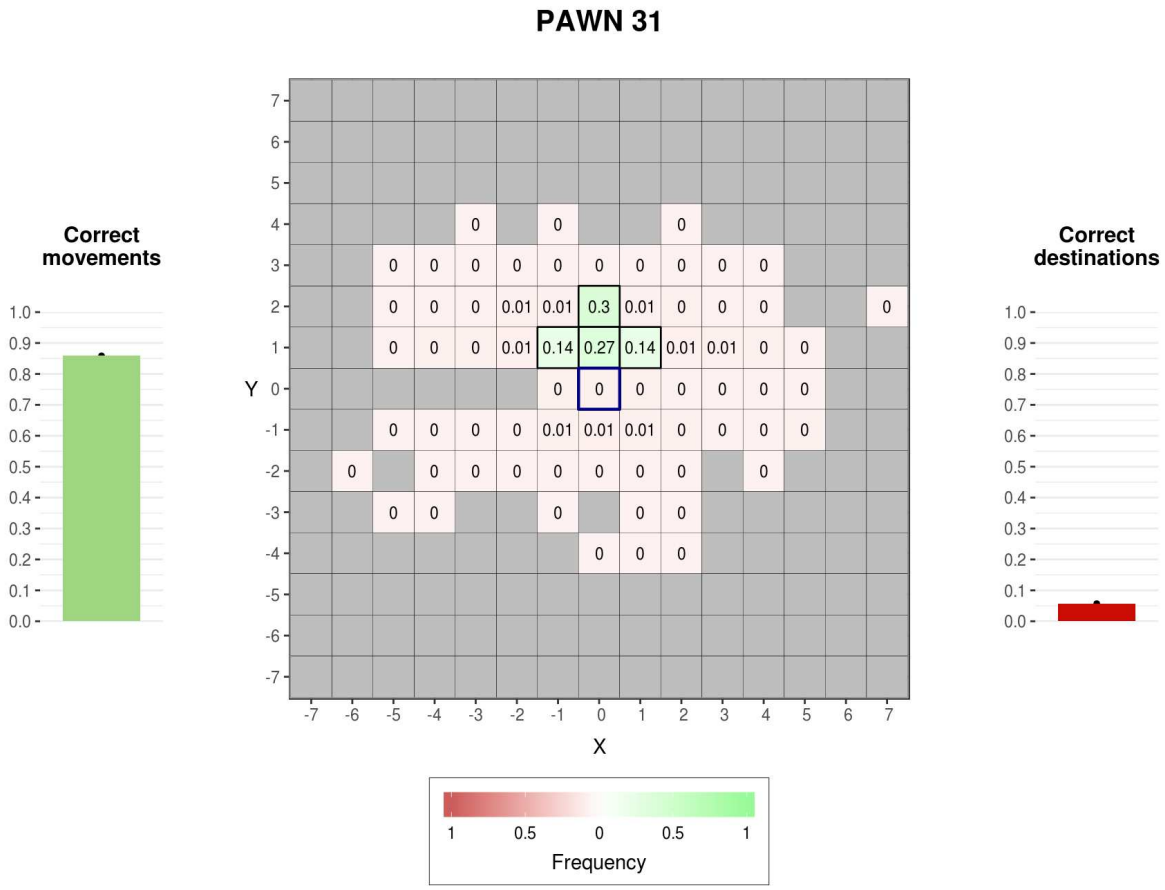


FIG. 38. This graph shows the R script schematic for Pawn at snapshot 31 and "less convolutional layers" model.

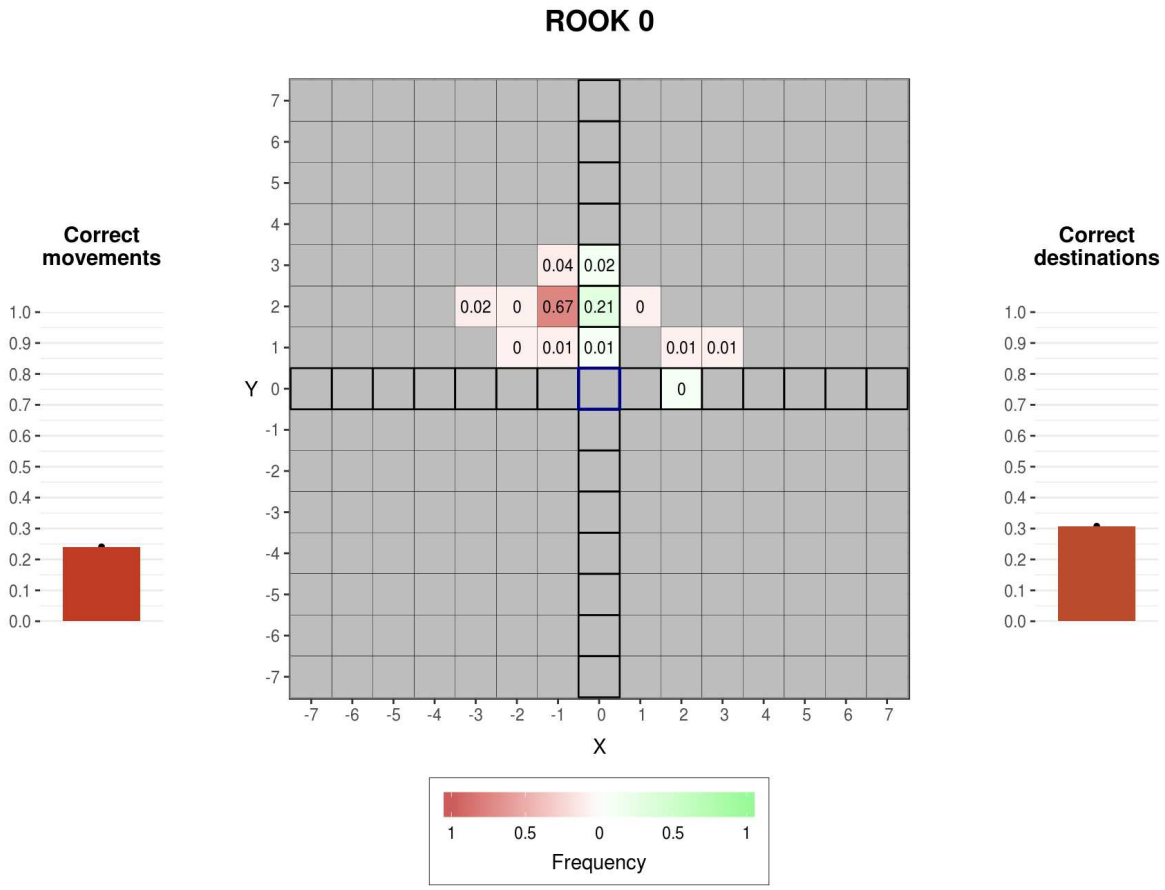


FIG. 39. This graph shows the R script schematic for Rook at snapshot 0 and "less convolutional layers" model.

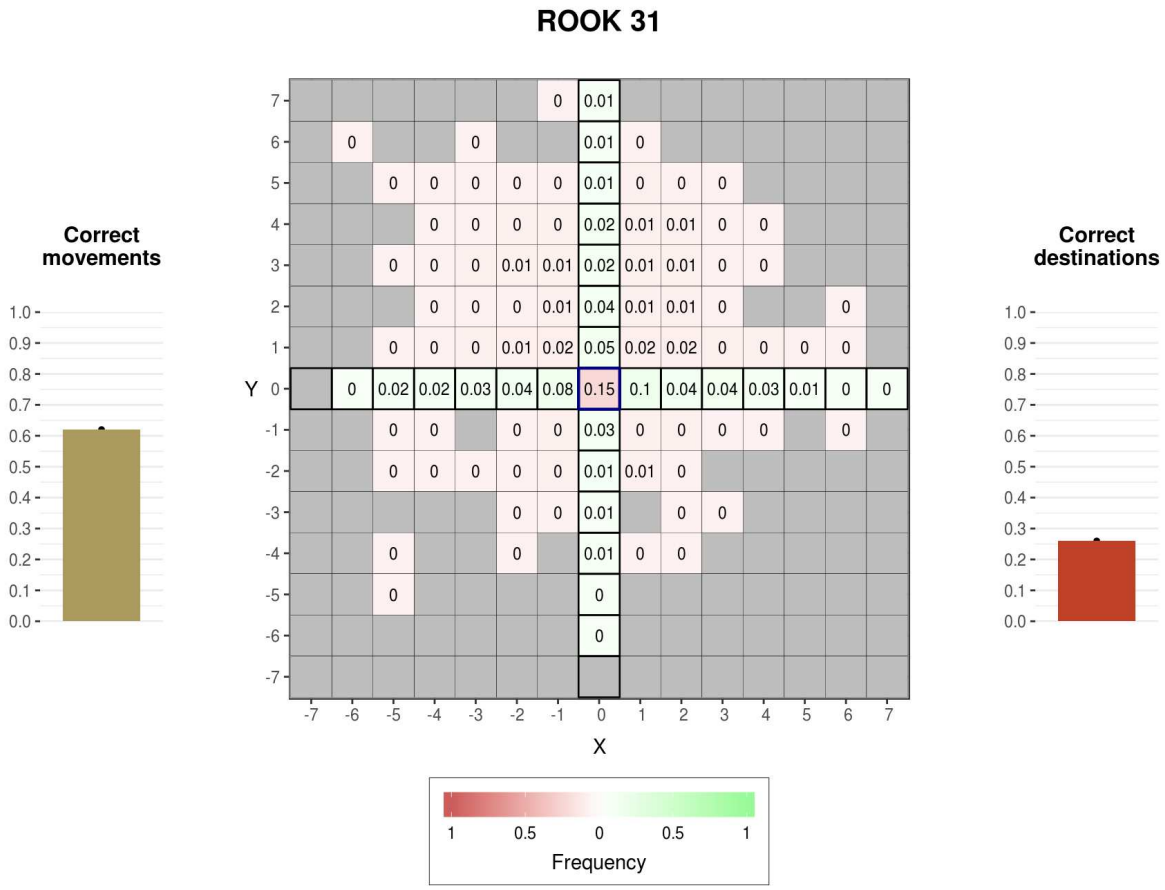


FIG. 40. This graph shows the R script schematic for Rook at snapshot 31 and "less convolutional layers" model.

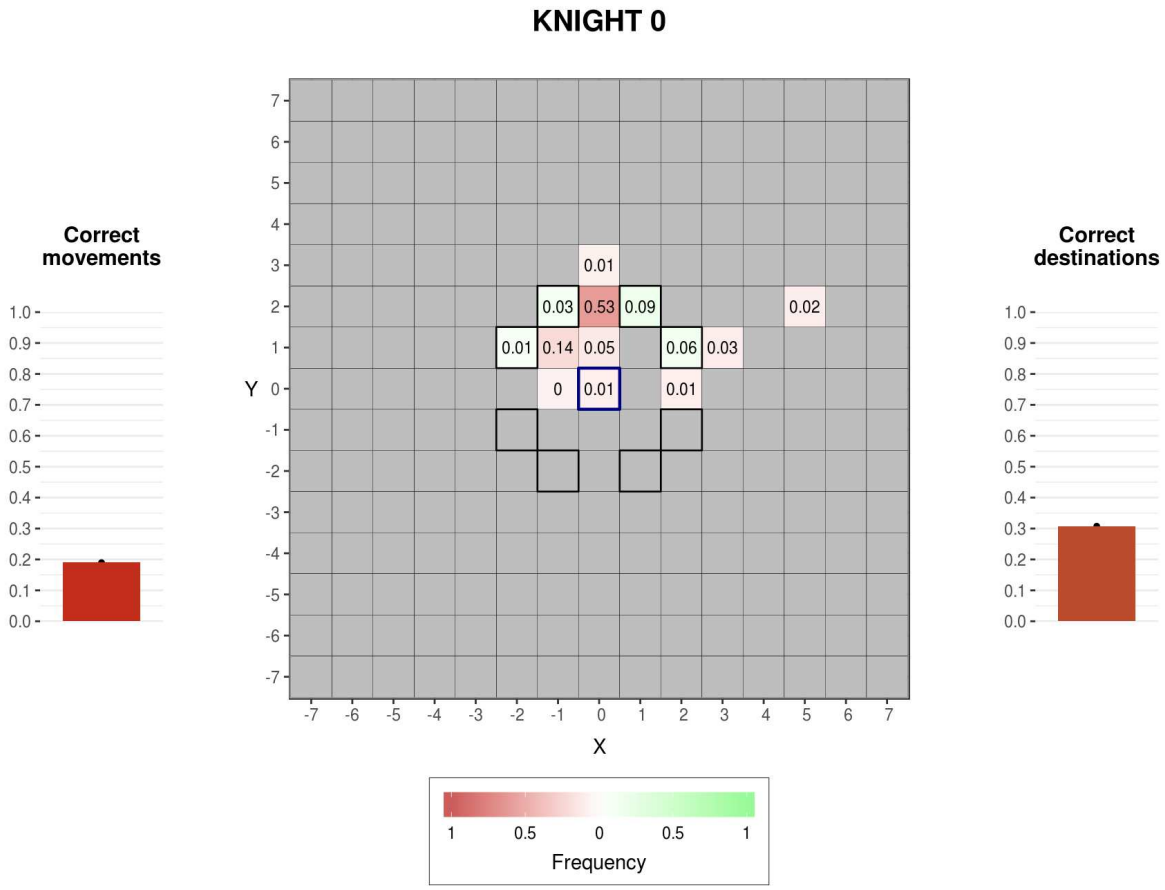


FIG. 41. This graph shows the R script schematic for Knight at snapshot 0 and "less convolutional layers" model.

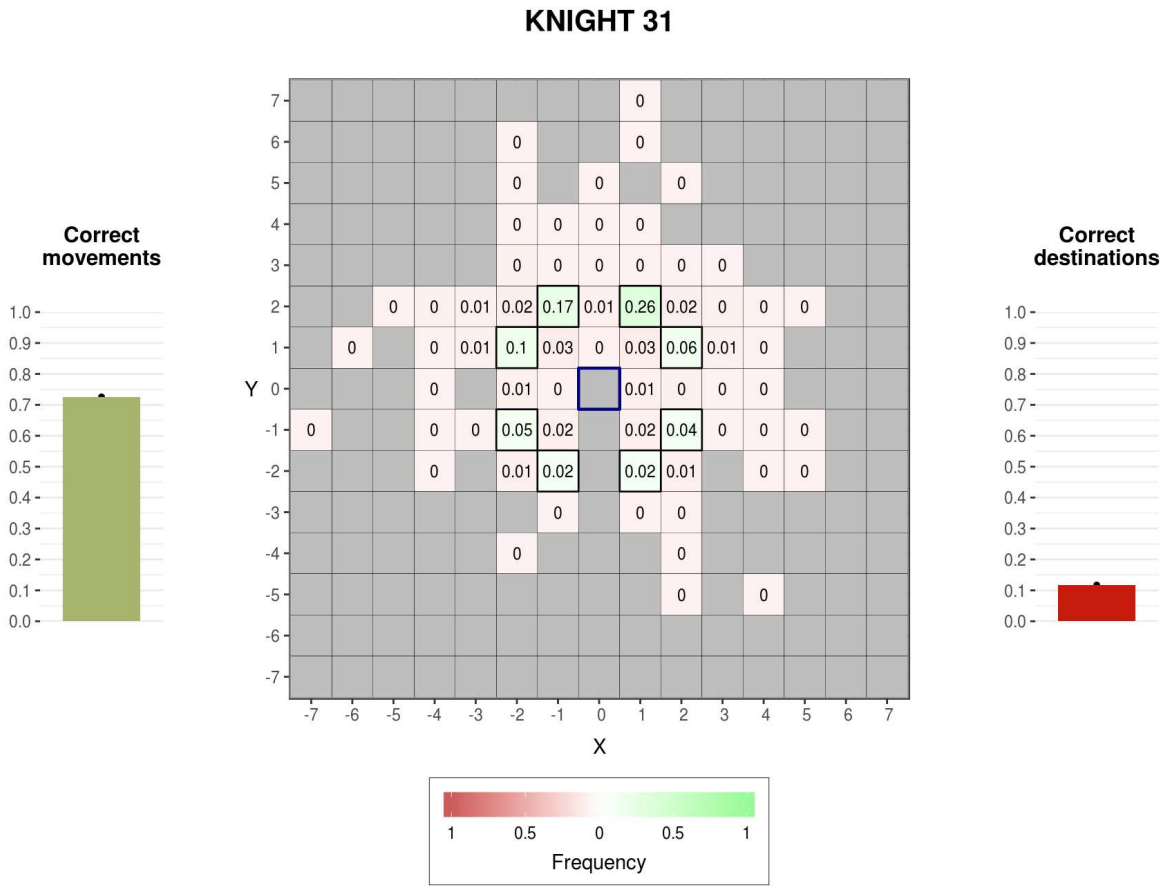


FIG. 42. This graph shows the R script schematic for Knight at snapshot 31 and "less convolutional layers" model.

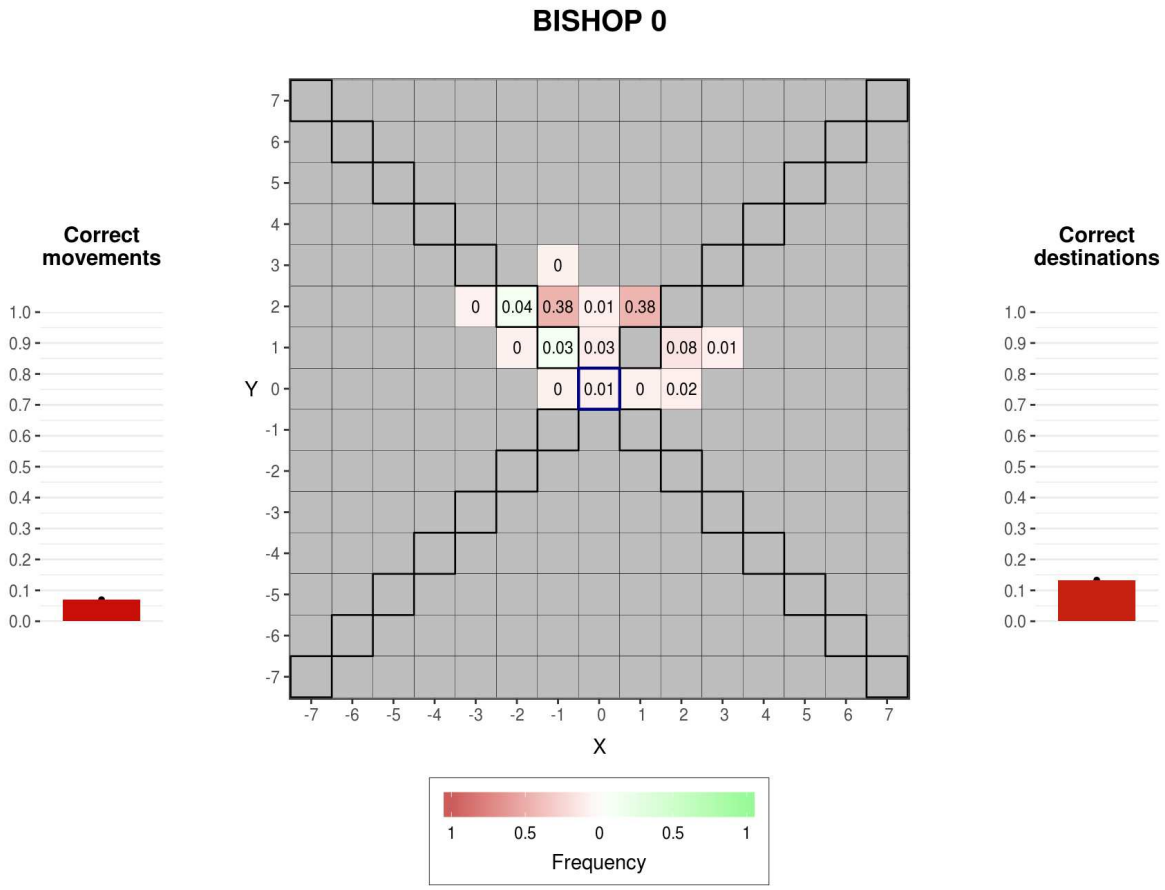


FIG. 43. This graph shows the R script schematic for Bishop at snapshot 0 and "less convolutional layers" model.

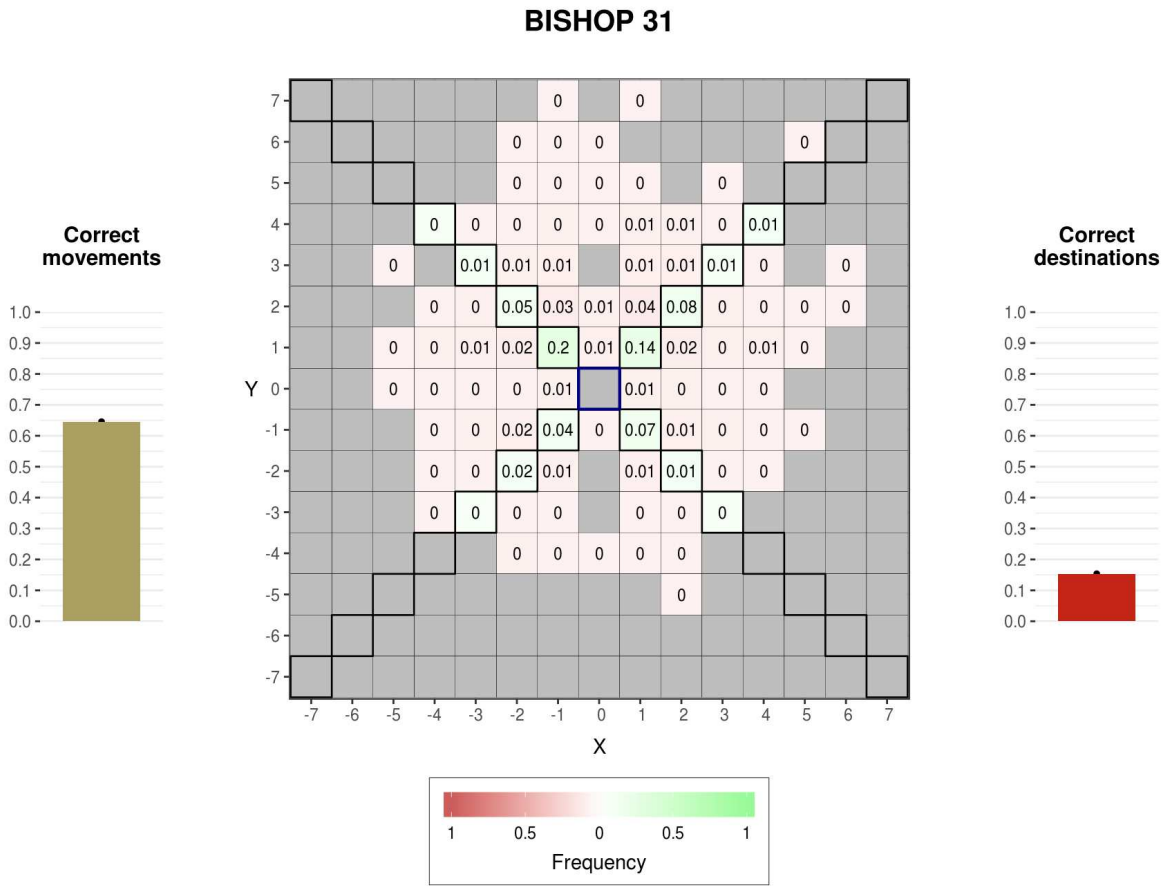


FIG. 44. This graph shows the R script schematic for Bishop at snapshot 31 and "less convolutional layers" model.

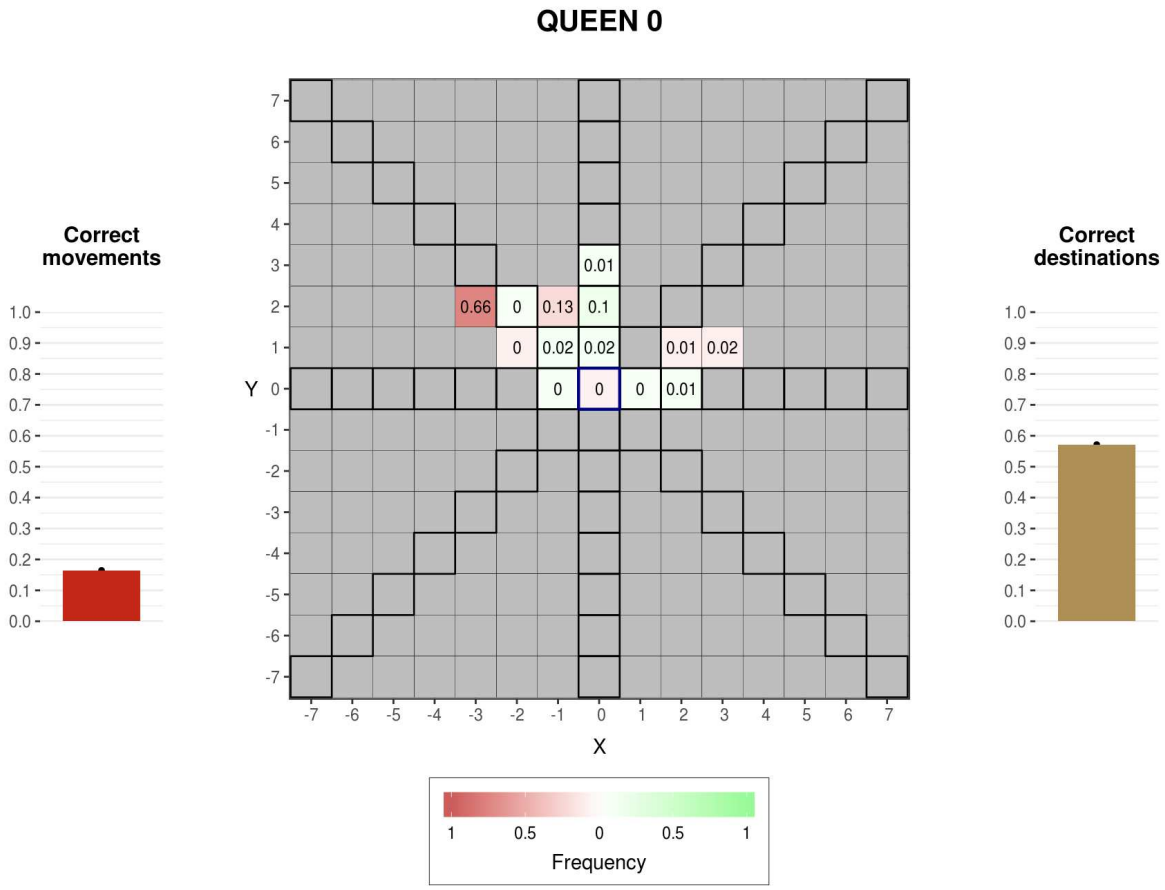


FIG. 45. This graph shows the R script schematic for Queen at snapshot 0 and "less convolutional layers" model.

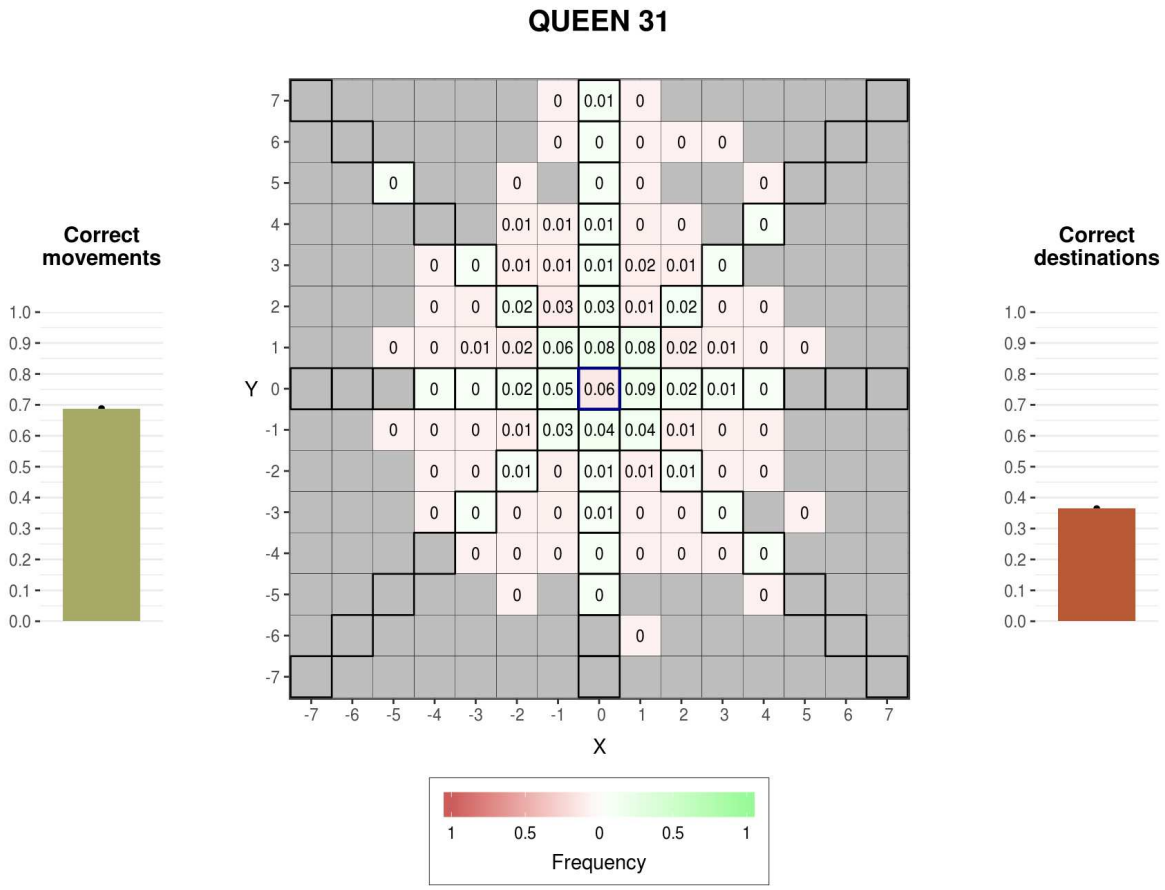


FIG. 46. This graph shows the R script schematic for Queen at snapshot 31 and "less convolutional layers" model.

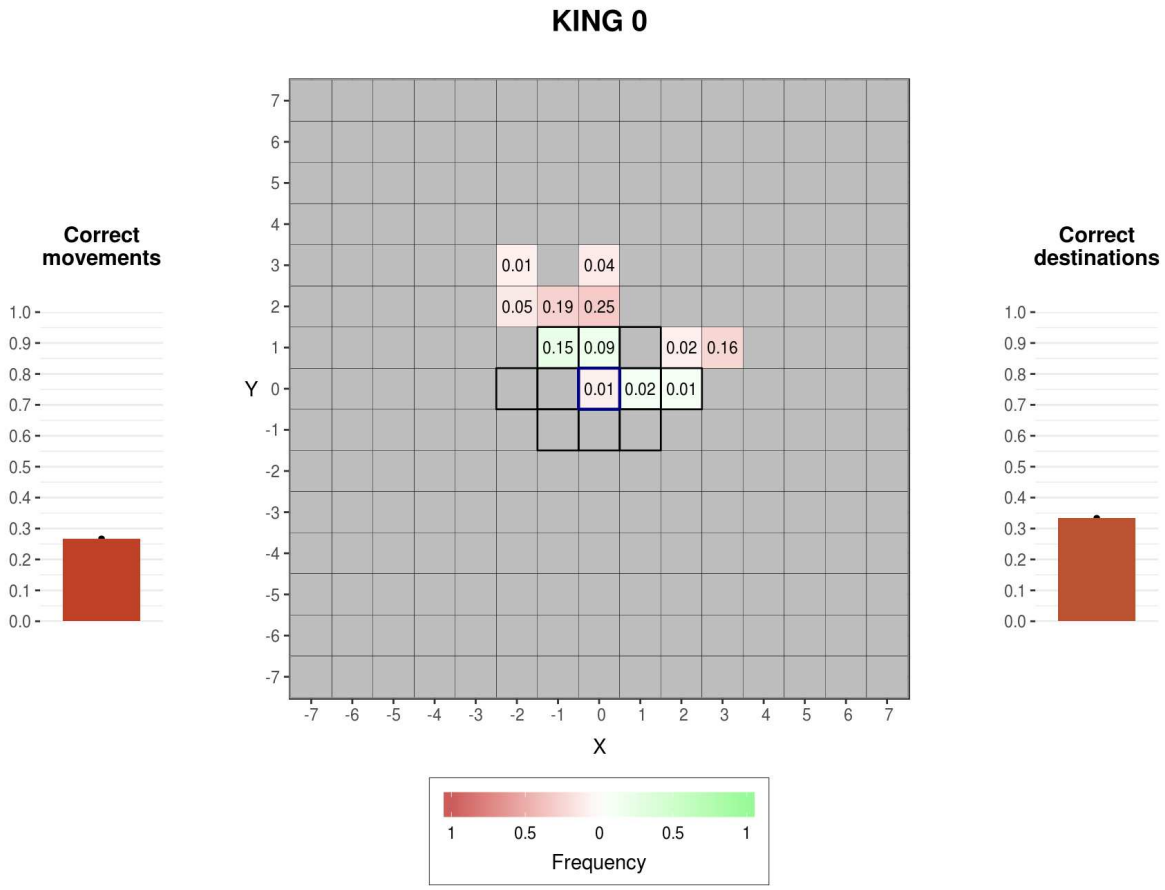


FIG. 47. This graph shows the R script schematic for King at snapshot 0 and "less convolutional layers" model.

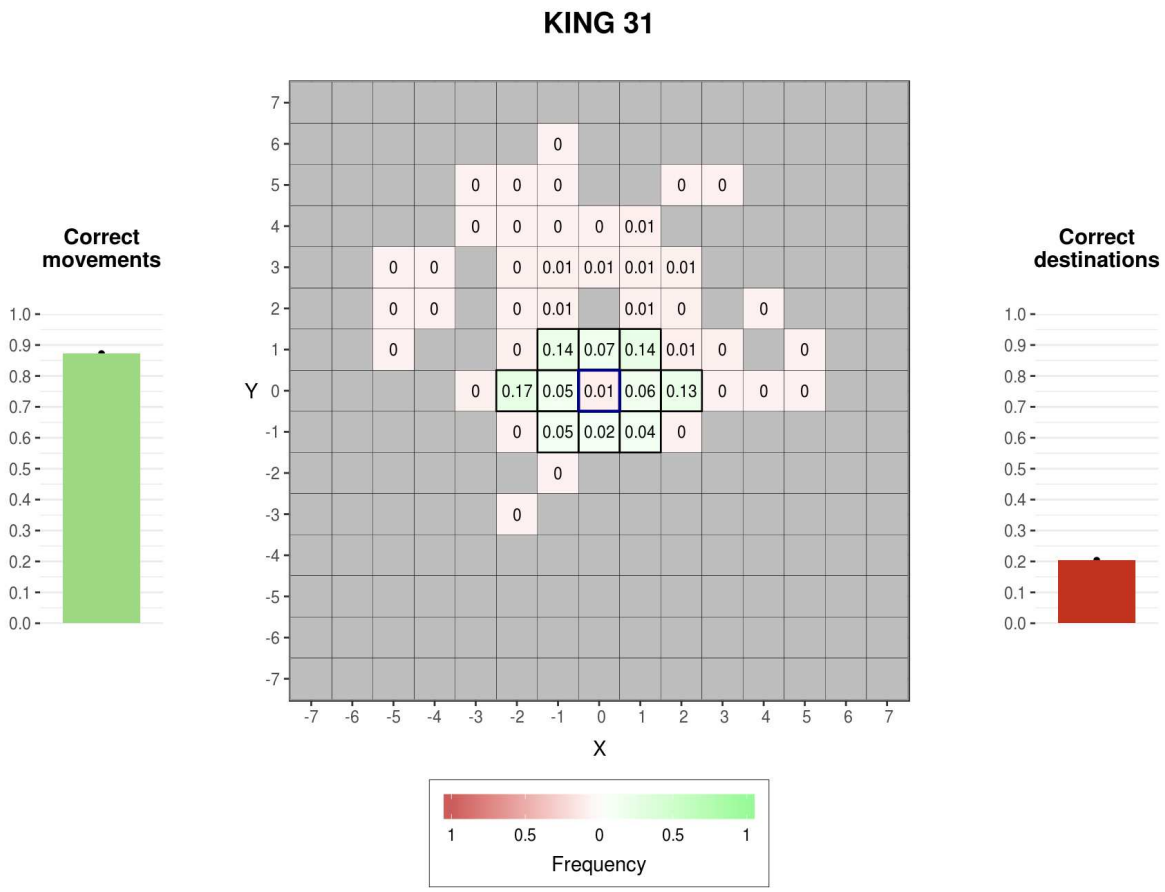


FIG. 48. This graph shows the R script schematic for King at snapshot 31 and "less convolutional layers" model.

Code

The following pages present the most important scripts used and programmed in this thesis. Because of the tabulation and line wrapping, some lines may not be visible in this document. The same code is available online at my Github profile.

Python scripts. This first scripts fed the data collected from the chess databases to the neural networks.

```
from ChessParser import ChessParser
import numpy as np
from keras.models import Sequential
from keras.losses import binary_crossentropy
from keras.callbacks import TensorBoard, Callback, ModelCheckpoint
from keras.backend.tensorflow_backend import set_session
from keras.initializers import glorot_normal
import keras.backend as K
import tensorflow as tf
from keras.layers import Conv2D, MaxPool2D, Flatten, Dropout,
    ↪ Dense
import os
import sys

try:
    file_path = sys.argv[1]
except IndexError:
    print "Usage: python cnn_template.py 'config_file'"
    exit()

if not file_path.startswith('/'):
    file_path = os.path.join(os.getcwd(), file_path)

# Prepare gpu
# Disable Tensorflow environments about CPU instructions
os.environ["CUDA_VISIBLE_DEVICES"]="0"
config = tf.ConfigProto()
# config.gpu_options.per_process_gpu_memory_fraction = 0.95
set_session(tf.Session(config = config))

# Variables
training_data_dir = "Databases/Chess/TrainData"
validation_data_dir = "Databases/Chess/TestData"

# Hyperparameters (read from config-file)
input_shape = (8, 8, 6)

hyperparameters = dict((x, [float(z) for z in y.split(',')]) if ',
    ↪ ' in y else (x, int(y)) if y.isdigit() else (x, y) for x, y
    ↪  in [tuple(x.strip().split()) for x in open(file_path, 'r')
    ↪ .readlines() if x.strip()])

name = str(hyperparameters['cnn_layers']) + 'cnn_'
```

```
for i in range(hyperparameters['cnn_layers']):
    name += str(int(hyperparameters['num_kernels_per_layer'][i]))
        ↪ + '-' + str(int(hyperparameters['kernel_sizes'][i])) +
        ↪ '_'

name += str(hyperparameters['pooling_layers']) + 'pooling_'
for i in range(hyperparameters['pooling_layers']):
    name += str(int(hyperparameters['pooling_layers_sizes'][i])) +
        ↪  '_'

name += str(hyperparameters['hdd_layers']) + 'hdd_'
for i in range(hyperparameters['hdd_layers']):
    name += str(int(hyperparameters['hidden_layers_sizes'][i])) +
        ↪ '_'

name += os.path.splitext(os.path.basename(file_path))[0]

if not os.path.exists('Models/%s' % name):
    os.makedirs('Models/%s' % name)

# First build CNN layers
model = Sequential()
cnn_index = 0
pool_index = 0
for i in range(hyperparameters['cnn_layers'] + hyperparameters['
    ↪ pooling_layers']):

    if i == 0:
        num_kernels = int(hyperparameters['num_kernels_per_layer'
            ↪ ][cnn_index])
        kernel_size = int(hyperparameters['kernel_sizes'][
            ↪ cnn_index])
        model.add(Conv2D(num_kernels, (int(kernel_size), int(
            ↪ kernel_size)), padding=hyperparameters['cnn_padding
            ↪ '], activation=hyperparameters['cnn_activation'],
            ↪ input_shape=input_shape, kernel_initializer='
            ↪ glorot_normal'))
        cnn_index += 1
    else:
        if i%(hyperparameters['cnn_layers']/4 + 1) + 1 ==
            ↪ hyperparameters['cnn_layers']/4 + 1:
            if pool_index == hyperparameters['pooling_layers'] -
                ↪ 1:
                model.add(MaxPool2D(pool_size=int(hyperparameters[
                    ↪ 'pooling_layers_sizes'][pool_index]),
                    ↪ data_format="channels_last"))
            else:
                model.add(MaxPool2D(pool_size=int(hyperparameters[
                    ↪ 'pooling_layers_sizes'][pool_index]),
                    ↪ strides=1, data_format="channels_last"))
            pool_index += 1
        else:
```

```
            num_kernels = int(hyperparameters['
                ↪ num_kernels_per_layer'][cnn_index])
            kernel_size = int(hyperparameters['kernel_sizes'][
                ↪ cnn_index])
            model.add(Conv2D(num_kernels, (int(kernel_size), int(
                ↪ kernel_size)), padding=hyperparameters['
                ↪ cnn_padding'], activation=hyperparameters['
                ↪ cnn_activation'], kernel_initializer='
                ↪ glorot_normal'))
            cnn_index += 1

# Add hidden layers
model.add(Flatten())
for i in range(hyperparameters['hdd_layers']):
    model.add(Dense(int(hyperparameters['hidden_layers_sizes'][i])
        ↪ , activation=hyperparameters['hdd_activation'],
        ↪ kernel_initializer='glorot_normal'))

def custom_accuracy(y_true, y_pred):
    return K.mean(tf.reduce_all(K.equal(y_true, K.round(y_pred)),
        ↪ axis=1))

model.compile(loss=binary_crossentropy, optimizer='sgd', metrics=[
    ↪ custom_accuracy])

# Save model configurations
def myprint(s):
    with open('Models/%s/summary' % name, 'a+') as f:
        f.write(s)
        f.write('\n')
model.summary(print_fn=myprint)
json = model.to_json()
with open('Models/%s/json' % name, 'w+') as f:
    f.write(json)

print "Generating validation data..."
validation_data_generator = ChessParser.cnn_chess_data_generator(
    ↪ validation_data_dir, validation=True, verbose=True)

validation_data_inp = np.empty((0, 8, 8, 6))
validation_data_out = np.empty((0, 32))

for game_input, game_output in validation_data_generator:
    validation_data_inp = np.concatenate((validation_data_inp,
        ↪ game_input))
    validation_data_out = np.concatenate((validation_data_out,
        ↪ game_output))

print "\nValidation data input/output shapes: " + str(
    ↪ validation_data_inp.shape) + " / " + str(
    ↪ validation_data_out.shape)
```

```
tensorboard_callback = TensorBoard(log_dir=os.path.join('logs/',
    ↪ name), histogram_freq=1, batch_size=32, write_graph=False,
    ↪ write_grads=True)
checkpoint_callback = ModelCheckpoint(filepath="Models/%s/model_{
    ↪ epoch:02d}_{val_loss:.2f}" % (name), monitor='val_loss',
    ↪ verbose=1, save_best_only=True, save_weights_only=False,
    ↪ mode="min", period=1)

model.fit_generator(generator=ChessParser.cnn_chess_data_generator
    ↪ (path=training_data_dir, validation=False, verbose=False),
                    steps_per_epoch=100000, epochs=1000000,
                        ↪ verbose=1, callbacks=[
                        ↪ tensorboard_callback,
                        ↪ checkpoint_callback],
                    validation_data=(validation_data_inp,
                        ↪ validation_data_out), validation_steps
                        ↪ =50, use_multiprocessing=True)
```

This second Python script contains the Chess Logic class.

```
import numpy as np


# noinspection SpellCheckingInspection
class ChessLogic(object):
    """
    Representation of a Chess Game in progress.
    Instance Variables:
            board:
                numpy array of size 2x7x8x8 (7 - one-hot vector of
                    ↪ figures)
                None    [1, 0, 0, 0, 0, 0, 0]
                Pawn    [0, 1, 0, 0, 0, 0, 0]
                Rook    [0, 0, 1, 0, 0, 0, 0]
                Knight  [0, 0, 0, 1, 0, 0, 0]
                Bishop  [0, 0, 0, 0, 1, 0, 0]
                Queen   [0, 0, 0, 0, 0, 1, 0]
                King    [0, 0, 0, 0, 0, 0, 1]
            white_en_passant:
                coordinates where black can kill white en passant
            black_en_passant:
                coordinates where white can kill black en passant
            black_can_castle:
                wether black can castle or not
            white_can_caste
                wether white can castle or not
            castle:
                Did the last player castle?
                    0 - No
                    1 - Small
                    2 - Large
            kill:
```

```
            Wether there was a kill or not in the last play
        current_player:
            False for Black, True for White
        promotion:
            Did the last player promote a pawn?
    """

    def __init__(self):
        self.board = np.zeros((2, 7, 8, 8))
        self.white_en_passant = None
        self.black_en_passant = None
        self.black_can_small_castle = True
        self.white_can_small_castle = True
        self.black_can_large_castle = True
        self.white_can_large_castle = True
        self.castle = 0
        self.kill = False
        self.current_player = True
        self.promotion = False

        self.prepare()

    def prepare(self):
        """
        Initialize the board position
        :return: None
        """

        white_board = self.board[1]
        black_board = self.board[0]

        # None Figures
        white_board[0][:6] = 1
        black_board[0][2:8] = 1
        # Pawns
        white_board[1][6] = 1
        black_board[1][1] = 1
        # Rooks
        white_board[2][7][0] = white_board[2][7][7] = 1
        black_board[2][0][0] = black_board[2][0][7] = 1
        # Knights
        white_board[3][7][1] = white_board[3][7][6] = 1
        black_board[3][0][1] = black_board[3][0][6] = 1
        # Bishops
        white_board[4][7][2] = white_board[4][7][5] = 1
        black_board[4][0][2] = black_board[4][0][5] = 1
        # Queens
        white_board[5][7][3] = 1
        black_board[5][0][3] = 1
        # Kings
        white_board[6][7][4] = 1
        black_board[6][0][4] = 1
```

```
    def move(self, initial_position, final_position,
        ↪ promotion_figure):
        """
        Move a player's figure to a desired position

        :param initial_position: (1x3 tuple) Figure's current
            ↪ position
        :param final_position: (1x3 tuple) Figure's final
            ↪ destination
        :param promotion_figure: (int) In case of promotion,
            ↪ figure to promote. Default queen
        :return: (bool) True if the movement was applied, False
            ↪ otherwise
        """

        # Get figure trying to move
        figure = ChessLogic.figure_for_player_at_position(self.
            ↪ board, self.current_player, initial_position)

        # If trying to move a None-Figure, False
        if not figure:
            return False

        # If movement is not valid, False
        if not self.is_valid_movement(figure, initial_position,
            ↪ final_position):
            return False

        # Reset en_passant varialble
        if self.white_en_passant and self.current_player == 0:
            self.white_en_passant = None
        elif self.black_en_passant and self.current_player == 1:
            self.black_en_passant = None

        # Update board
        self.board[int(self.current_player)][0][initial_position]
            ↪ = 1
        self.board[int(self.current_player)][0][final_position] =
            ↪ 0
        self.board[int(self.current_player)][figure][
            ↪ initial_position] = 0
        self.board[int(self.current_player)][figure][
            ↪ final_position] = 1

        # If the movement implied a promotion, change board
            ↪ position accordingly
        if self.promotion:
            self.board[int(self.current_player)][promotion_figure
                ↪ ][final_position] = 1
            self.board[int(self.current_player)][figure][
                ↪ final_position] = 0
            self.promotion = False
        # Else if movement implied a kill
```

```
        if self.kill:
            killed_figure = ChessLogic.
                ↪ figure_for_player_at_position(self.board, not
                ↪ self.current_player, final_position)
            self.board[int(not self.current_player)][0][
                ↪ final_position] = 1
            self.board[int(not self.current_player)][killed_figure
                ↪ ][final_position] = 0
            self.kill = False
        # Else if small castle
        elif self.castle == 1:
            x = 7 if self.current_player else 0

            self.board[int(self.current_player)][0][(x, 7)] = 1
            self.board[int(self.current_player)][0][(x, 5)] = 0
            self.board[int(self.current_player)][2][(x, 7)] = 0
            self.board[int(self.current_player)][2][(x, 5)] = 1

            self.castle = 0
        # Else if large castle
        elif self.castle == 2:
            x = 7 if self.current_player else 0

            self.board[int(self.current_player)][0][(x, 0)] = 1
            self.board[int(self.current_player)][0][(x, 3)] = 0
            self.board[int(self.current_player)][2][(x, 0)] = 0
            self.board[int(self.current_player)][2][(x, 3)] = 1

            self.castle = 0

        # Change current Player
        self.current_player = not self.current_player

        return True

    def is_valid_movement(self, figure, initial_pos, final_pos):
        """
        Given a figure and an initial position, returns if it's
            ↪ possible for the figure to move to the final
            ↪ position

        :param figure: (int) Figure to move {P: 1, R: 2, N: 3, B:
            ↪ 4, Q: 5, K: 6}
        :param initial_pos: (1x3 tuple) initial position
        :param final_pos: (1x3 tuple) final position
        :return: (bool) True if movement is valid, False otherwise
        """

        # If trying to move a non-figure position, False
        if not figure:
            return False
        # If trying to move outbounds, False
```

```
        if (np.array(final_pos) < 0).any() or (np.array(final_pos)
            ↪  > 7).any():
            return False

        # Get straight and diagonal ranges for movement
        straight_range = self.straight_coordinate_range(
            ↪ initial_pos, final_pos)
        diagonal_range = self.diagonal_coordinate_range(
            ↪ initial_pos, final_pos)

        # Pawn
        if figure == 1:
            # If trying to move forward
            if straight_range.size and ((final_pos[0] >
                ↪ initial_pos[0] and self.current_player == 0) or
                                        (final_pos[0] <
                                            ↪ initial_pos[0] and
                                            ↪ self.current_player
                                            ↪  == 1)):
                # If trying to move one step forward
                if straight_range.shape[0] == 1:
                    # Check if position is empty
                    if self.is_position_empty(final_pos):
                        # If reaching last rank, promotion
                        if final_pos[0] == 7*(not self.
                            ↪ current_player):
                            self.promotion = True
                        return True
                    return False
                # Else if trying to move two steps forward
                elif straight_range.shape[0] == 2:
                    # If not in second row, False
                    if (initial_pos[0] != 1 and self.
                        ↪ current_player == 0) or (self.
                        ↪ current_player == 1 and initial_pos[0]
                        ↪ != 6):
                        return False
                    for coord in straight_range:
                        if not self.is_position_empty(coord):
                            return False
                    # If movement is correct, there is a
                        ↪ possibility that an en passant kill
                        ↪ takes place
                    if self.current_player == 1:
                        self.white_en_passant = list(
                            ↪ straight_range[0])
                    else:
                        self.black_en_passant = list(
                            ↪ straight_range[0])
                    return True
                # Else (trying to move more than 2 steps forward)
                return False
            # Else trying to kill
```

```
            elif diagonal_range.shape[0] == 1 and ((final_pos[0] >
                ↪  initial_pos[0] and self.current_player == 0)
                ↪ or
                                                    (final_pos[0] <
                                                        ↪
                                                        ↪ initial_pos
                                                        ↪ [0] and
                                                        ↪ self.
                                                        ↪ current_player
                                                        ↪  == 1)):
                # If currently playing white and black_en_passant
                    ↪ is final_pos, kill
                if self.current_player == 1 and np.array_equal(
                    ↪ self.black_en_passant, final_pos):
                    # In order to kill properly, we need to move
                        ↪ back the enemy pawn to final_pos
                    # We need to change board position
                    self.board[int(not self.current_player)][0][
                        ↪ final_pos] = 0
                    self.board[int(not self.current_player)][0][
                        ↪ final_pos[0] + 1][final_pos[1]] = 1
                    self.board[int(not self.current_player)][1][
                        ↪ final_pos] = 1
                    self.board[int(not self.current_player)][1][
                        ↪ final_pos[0] + 1][final_pos[1]] = 0

                    self.kill = True
                    return True
                # Else if currently playing black and
                    ↪ white_en_passant is final_pos, kill
                elif self.current_player == 0 and np.array_equal(
                    ↪ self.white_en_passant, final_pos):
                    # In order to kill properly, we need to move
                        ↪ back the enemy pawn to final_pos
                    # We need to change board position
                    self.board[int(not self.current_player)][0][
                        ↪ final_pos] = 0
                    self.board[int(not self.current_player)][0][
                        ↪ final_pos[0] - 1][final_pos[1]] = 1
                    self.board[int(not self.current_player)][1][
                        ↪ final_pos] = 1
                    self.board[int(not self.current_player)][1][
                        ↪ final_pos[0] - 1][final_pos[1]] = 0

                    self.kill = True
                    return True
                # Else if enemy figure in final_pos, kill
                elif ChessLogic.figure_for_player_at_position(self
                    ↪ .board, not self.current_player, final_pos)
                    ↪  != 0:
                    # If killing in last line, promotion!
```

```
                if self.current_player == 1 and final_pos[0]
                    ↪ == 0 or self.current_player == 0 and
                    ↪ final_pos[0] == 7:
                    self.promotion = True
                self.kill = True
                return True
            # Pawn not capable of killing
            return False

        # Else (not a valid move for pawn)
        return False

    # Rook
    elif figure == 2:
        # If trying to move to same position, False
        if not straight_range.size:
            return False
        # Else if no figures in the way
        for coord in straight_range[:-1]:
            if not self.is_position_empty(coord):
                return False
        # If there is an ally figure at final_pos, False
        if ChessLogic.figure_for_player_at_position(self.board
            ↪ , self.current_player, final_pos) != 0:
            return False
        # If code execution reaches this point, the movement
            ↪ is valid, so player cannot castle
        if self.current_player == 1:
            # If moving left rook, player cannot large castle
            if np.array_equal(initial_pos, (7, 0)):
                self.white_can_large_castle = False
            # Else if moving right rook, player cannot small
                ↪ castle
            elif np.array_equal(initial_pos, (7, 7)):
                self.white_can_small_castle = False

        elif self.current_player == 0:
            # If moving left rook, player cannot large castle
            if np.array_equal(initial_pos, (0, 0)):
                self.black_can_large_castle = False
            # Else if moving right rook, player cannot small
                ↪ castle
            elif np.array_equal(initial_pos, (0, 7)):
                self.black_can_small_castle = False

        # If there is an enemy figure at final_pos, kill
        if ChessLogic.figure_for_player_at_position(self.board
            ↪ , not self.current_player, final_pos) != 0:
            self.kill = True
        return True

    # Knight
    elif figure == 3:
```

```
            # Check if movement is valid for knight
            if (abs(final_pos[0] - initial_pos[0]) == 2 and abs(
                ↪ final_pos[1] - initial_pos[1]) == 1) or\
                    (abs(final_pos[0] - initial_pos[0]) == 1 and
                        ↪ abs(final_pos[1] - initial_pos[1]) ==
                        ↪ 2):
                # If there is an ally figure at final_pos, False
                if ChessLogic.figure_for_player_at_position(self.
                    ↪ board, self.current_player, final_pos) !=
                    ↪ 0:
                    return False
                # Else if there is an enemy figure at final_pos,
                    ↪ kill
                if ChessLogic.figure_for_player_at_position(self.
                    ↪ board, not self.current_player, final_pos)
                    ↪ != 0:
                    self.kill = True
                return True
            # Else (not valid move for Knight)
            return False

        # Bishop
        elif figure == 4:
            # If trying to move to same position, False
            if not diagonal_range.size:
                return False
            # Else if no figures in the way
            for coord in diagonal_range[:-1]:
                if not self.is_position_empty(coord):
                    return False
            # If there is an ally figure at final_pos, False
            if ChessLogic.figure_for_player_at_position(self.board
                ↪ , self.current_player, final_pos) != 0:
                return False
            # If code execution reaches this point, the movement
                ↪ is valid, ergo True
            # If there is an enemy figure at final_pos, kill
            elif ChessLogic.figure_for_player_at_position(self.
                ↪ board, not self.current_player, final_pos) !=
                ↪ 0:
                self.kill = True
            return True

        # Queen
        elif figure == 5:
            # If trying to move in a straight line
            if straight_range.size:
                for coord in straight_range[:-1]:
                    if not self.is_position_empty(coord):
                        return False
                # If there is an ally figure at final_pos, False
```

```
                if ChessLogic.figure_for_player_at_position(self.
                    ↪ board, self.current_player, final_pos) !=
                    ↪ 0:
                    return False
                # If code execution reaches this point, the
                    ↪ movement is valid, ergo True
                # If there is an enemy figure at final_pos, kill
                elif ChessLogic.figure_for_player_at_position(self
                    ↪ .board, not self.current_player, final_pos)
                    ↪  != 0:
                    self.kill = True
                return True

            # Else if trying to move in a diagonal line
            elif diagonal_range.size:
                for coord in diagonal_range[:-1]:
                    if not self.is_position_empty(coord):
                        return False
                # If there is an ally figure at final_pos, False
                if ChessLogic.figure_for_player_at_position(self.
                    ↪ board, self.current_player, final_pos) !=
                    ↪ 0:
                    return False
                # If code execution reaches this point, the
                    ↪ movement is valid, ergo True
                # If there is an enemy figure at final_pos, kill
                elif ChessLogic.figure_for_player_at_position(self
                    ↪ .board, not self.current_player, final_pos)
                    ↪  != 0:
                    self.kill = True
                return True
            # Else (not a valid movement for Queen)
            return False

        # King
        elif figure == 6:
            # If trying to move one straight or diagonal step
            if straight_range.shape[0] == 1 or diagonal_range.
                ↪ shape[0] == 1:
                # If there is an ally figure at final_pos, False
                if ChessLogic.figure_for_player_at_position(self.
                    ↪ board, self.current_player, final_pos) !=
                    ↪ 0:
                    return False
                else:
                    # If there is an enemy figure at final_pos,
                        ↪ kill
                    if ChessLogic.figure_for_player_at_position(
                        ↪ self.board, not self.current_player,
                        ↪ final_pos) != 0:
                        self.kill = True
                    # Movement is valid, but player cannot castle
                        ↪ anymore
```

```
                    if self.current_player == 1:
                        self.white_can_small_castle = False
                        self.white_can_large_castle = False
                    else:
                        self.black_can_small_castle = False
                        self.black_can_large_castle = False
                    return True
            # Else if trying to large castle
            elif ((np.array_equal(final_pos, [0, 2]) and self.
                ↪ current_player == 0) or
                  (np.array_equal(final_pos, [7, 2]) and self.
                      ↪ current_player == 1)):
                # If player cannot castle, False
                if self.current_player == 1 and not self.
                    ↪ white_can_large_castle:
                    return False
                # If player cannot castle, False
                if self.current_player == 0 and not self.
                    ↪ black_can_large_castle:
                    return False

                # If figures in the way, return False
                for coord in straight_range:
                    if not self.is_position_empty(coord):
                        return False
                self.castle = 2
                return True
            # Else if trying to small castle
            elif ((np.array_equal(final_pos, [0, 6]) and self.
                ↪ current_player == 0) or
                  (np.array_equal(final_pos, [7, 6]) and self.
                      ↪ current_player == 1)):
                # If player cannot castle, False
                if self.current_player == 1 and not self.
                    ↪ white_can_small_castle:
                    return False
                # If player cannot castle, False
                if self.current_player == 0 and not self.
                    ↪ black_can_small_castle:
                    return False

                # If figures in the way, return False
                for coord in straight_range:
                    if not self.is_position_empty(coord):
                        return False
                self.castle = 1
                return True
            return False

    def is_position_empty(self, coord):
        """
        Given a coordinate, returns True if the position is empty
            ↪ for both players, False otherwise
```

```
        :param coord: (1x2 tuple) Coordinate to check if empty
        :return: (bool) True if its empty, False otherwise
        """
        white_figure = ChessLogic.figure_for_player_at_position(
            ↪ self.board, 1, coord)
        black_figure = ChessLogic.figure_for_player_at_position(
            ↪ self.board, 0, coord)
        return white_figure == black_figure == 0

    @staticmethod
    def figure_for_player_at_position(board, player, coord):
        """
        Given a player and a coordinate, returns the figure.
        :param player: (int) The player
        :param coord: (1x2 tuple) The position in board
        :return:
        """
        position = board[int(player)].T[coord[1]][coord[0]]
        return np.where(position == 1)[0][0]

    def
        ↪ will_player_be_in_check_after_removing_figure_at_location_to_location
        ↪ (self, ori, fin):
        # First, assure ori and fin are tuples
        ori = tuple(ori)
        fin = tuple(fin)

        king_board = self.board[int(self.current_player)][6]
        king_where = np.where(king_board == 1)
        king_coord = (king_where[0][0], king_where[1][0])

        # Apply changes on a temporary board
        temp_board = np.copy(self.board)
        moving_figure = ChessLogic.figure_for_player_at_position(
            ↪ self.board, self.current_player, ori)
        temp_board[int(self.current_player)][moving_figure][ori] =
            ↪ 0
        temp_board[int(self.current_player)][moving_figure][fin] =
            ↪ 1
        temp_board[int(self.current_player)][0][ori] = 1
        temp_board[int(self.current_player)][0][fin] = 0

        # If it implies a kill
        enemy_figure = ChessLogic.figure_for_player_at_position(
            ↪ self.board, not self.current_player, fin)
        if enemy_figure != 0:
            temp_board[int(not self.current_player)][enemy_figure
                ↪ ][fin] = 0
            temp_board[int(not self.current_player)][0][fin] = 1

        diagonals = ChessLogic.diagonal_coordinate_range(
            ↪ king_coord, ori)
```

```
        straights = ChessLogic.straight_coordinate_range(
            ↪ king_coord, ori)

        if diagonals.size:
            killing_figures = [4, 5]
            if ori[0] < king_coord[0] and ori[1] < king_coord[1]:
                diag = np.array(ori) - min(ori)
            elif ori[0] < king_coord[0] and ori[1] > king_coord
                ↪ [1]:
                diag = (ori[0], ori[1])
                while diag[0] > 0 and diag[1] < 7:
                    diag = (diag[0] - 1, diag[1] + 1)
            elif ori[0] > king_coord[0] and ori[1] < king_coord
                ↪ [1]:
                diag = (ori[0], ori[1])
                while diag[0] < 7 and diag[1] > 0:
                    diag = (diag[0] + 1, diag[1] - 1)
            else:
                diag = np.array(ori) + (7 - max(ori))
            all_diagonals = ChessLogic.diagonal_coordinate_range(
                ↪ king_coord, diag)
            for d in all_diagonals:
                ally_fig = np.where(temp_board[int(self.
                    ↪ current_player)].T[d[1]][d[0]] == 1)[0][0]
                if ally_fig != 0:
                    return False
                enemy_fig = np.where(temp_board[int(not self.
                    ↪ current_player)].T[d[1]][d[0]] == 1)[0][0]
                if enemy_fig in killing_figures:
                    return True
                elif enemy_fig != 0:
                    return False

        elif straights.size:
            killing_figures = [2, 5]
            if ori[0] < king_coord[0]:
                line = (0, king_coord[1])
            elif ori[0] > king_coord[0]:
                line = (7, king_coord[1])
            elif ori[1] < king_coord[1]:
                line = (king_coord[0], 0)
            else:
                line = (king_coord[0], 7)
            all_lines = ChessLogic.straight_coordinate_range(
                ↪ king_coord, line)
            for l in all_lines:
                if np.where(temp_board[int(self.current_player)].T
                    ↪ [l[1]][l[0]] == 1)[0][0] != 0:
                    return False
                f = np.where(temp_board[int(not self.
                    ↪ current_player)].T[l[1]][l[0]] == 1)[0][0]
                if f in killing_figures:
                    return True
```

```
                elif f != 0:
                    return False

        return False

    @staticmethod
    def straight_coordinate_range(initial_pos, final_pos):
        """
        Given an initial 2d tuple and a final 2d tuple, return all straight coordinates between both.
        :param initial_pos: (1x2 tuple) Initial position
        :param final_pos: (1x2 tuple) Final position
        :return: (np.array) of coordinates
        """

        delta_x = final_pos[0] - initial_pos[0]
        delta_y = final_pos[1] - initial_pos[1]

        # If not a straight coords, False
        if delta_x == delta_y or (delta_x != 0 and delta_y != 0):
            return np.array([])

        # If straight in x
        elif delta_x == 0:
            return np.array([[initial_pos[0], x] for x in range(initial_pos[1] + np.sign(delta_y), final_pos[1] + np.sign(delta_y), np.sign(delta_y))])

        # Else if straight in y)
        elif delta_y == 0:
            return np.array([[x, initial_pos[1]] for x in range(initial_pos[0] + np.sign(delta_x),
```

```
                                                                    final_pos
                                                                        ↪ [0]
                                                                        ↪
                                                                        ↪ +
                                                                        ↪
                                                                        ↪ np
                                                                        ↪ .
                                                                        ↪ sign
                                                                        ↪ (
                                                                        ↪ delta_x
                                                                        ↪ )
                                                                        ↪ ,
                                                                        ↪
                                                                    np
                                                                        ↪ .
                                                                        ↪ sign
                                                                        ↪ (
                                                                        ↪ delta_x
                                                                        ↪ )
                                                                        ↪ )
                                                                        ↪ ])
                                                                        ↪

    @staticmethod
    def diagonal_coordinate_range(initial_pos, final_pos):
        """
        Given an initial 2d tuple and a final 2d tuple, return all
            ↪  diagonal coordinates between both.
        :param initial_pos: (1x2 tuple) Initial position
        :param final_pos: (1x2 tuple) Final position
        :return: (np.array) of coordinates
        """
        delta_x = final_pos[0] - initial_pos[0]
        delta_y = final_pos[1] - initial_pos[1]

        # If not diagonal coords, False
        if abs(delta_x) != abs(delta_y) or (delta_x == 0 and
            ↪ delta_y == 0):
            return np.array([])

        return np.array(zip([x for x in range(initial_pos[0] + np.
            ↪ sign(delta_x),
                                              final_pos[0] + np.
                                                  ↪ sign(delta_x)
                                                  ↪ ,
                                              np.sign(delta_x))],
                            [x for x in range(initial_pos[1] + np.
                                ↪ sign(delta_y),
                                              final_pos[1] + np.
                                                  ↪ sign(delta_y)
                                                  ↪ ,
                                              np.sign(delta_y))]))
```

```
    @staticmethod
    def are_coordinates_straight(coord_a, coord_b):
        if coord_a[0] == coord_b[0] or coord_a[1] == coord_b[1]:
            return True

        return False

    @staticmethod
    def are_coordinates_diagonal(coord_a, coord_b):
        delta_x = coord_b[0] - coord_a[0]
        delta_y = coord_b[1] - coord_a[1]

        if abs(delta_x) != abs(delta_y) or (delta_x == 0 and
            ↪ delta_y == 0):
            return False

        return True

    @staticmethod
    def valid_knight_positions_for_knight_at_location(coord):
        up_right = (coord[0] - 2, coord[1] + 1)
        right_up = (coord[0] - 1, coord[1] + 2)
        right_down = (coord[0] + 1, coord[1] + 2)
        down_right = (coord[0] + 2, coord[1] + 1)
        down_left = (coord[0] + 2, coord[1] - 1)
        left_down = (coord[0] + 1, coord[1] - 2)
        left_up = (coord[0] - 1, coord[1] - 2)
        up_left = (coord[0] - 2, coord[1] - 1)

        possible_positions = [up_right, right_up, right_down,
            ↪ down_right, down_left, left_down, left_up, up_left]
        valid_positions = []

        for position in possible_positions:
            if not (np.array(position) < 0).any() and not (np.
                ↪ array(position) > 7).any():
                valid_positions.append(position)

        return valid_positions

    def valid_bishop_positions_for_bishop_at_location(self, coord)
        ↪ :
        valid_coordinates = []
        np_coord = np.array(coord)

        # Upper Left
        ul = ChessLogic.diagonal_coordinate_range(coord, np_coord
            ↪ - min(coord))
        for pos in ul:
            valid_coordinates.append(pos)
            if not self.is_position_empty(pos):
                break
```

```
        # Lower Right
        lr = ChessLogic.diagonal_coordinate_range(coord, np_coord
            ↪ + (7 - max(np_coord)))
        for pos in lr:
            valid_coordinates.append(pos)
            if not self.is_position_empty(pos):
                break

        # Upper Right
        x = coord[1] + 1
        y = coord[0] - 1
        while x < 8 and y >= 0:
            valid_coordinates.append(np.array((y, x)))
            if not self.is_position_empty((y, x)):
                break
            x += 1
            y -= 1

        # Lower Left
        x = coord[1] - 1
        y = coord[0] + 1
        while x >= 0 and y < 8:
            valid_coordinates.append(np.array((y, x)))
            if not self.is_position_empty((y, x)):
                break
            x -= 1
            y += 1

        return valid_coordinates

    @staticmethod
    def valid_diagonal_pawn_positions_for_pawn_at_location(coord):
        valid_coordiantes = []

        # Upper Left
        y = coord[0] - 1
        x = coord[1] - 1
        if 0 <= y < 8 and 0 <= x < 8:
            valid_coordiantes.append((y, x))

        # Upper Right
        y = coord[0] - 1
        x = coord[1] + 1
        if 0 <= y < 8 and 0 <= x < 8:
            valid_coordiantes.append((y, x))

        # Lower Right
        y = coord[0] + 1
        x = coord[1] + 1
        if 0 <= y < 8 and 0 <= x < 8:
            valid_coordiantes.append((y, x))

        # Lower Left
```

```
        y = coord[0] + 1
        x = coord[1] - 1
        if 0 <= y < 8 and 0 <= x < 8:
            valid_coordiantes.append((y, x))

        return valid_coordiantes

    @staticmethod
    def valid_king_straight_positions_for_king_at_location(coord):
        valid_coordiantes = []

        # Up
        y = coord[0] - 1
        if y >= 0:
            valid_coordiantes.append((y, coord[1]))

        # Right
        x = coord[1] + 1
        if x < 8:
            valid_coordiantes.append((coord[0], x))

        # Down
        y = coord[0] + 1
        if y < 8:
            valid_coordiantes.append((y, coord[1]))

        # Left
        x = coord[1] - 1
        if x >= 0:
            valid_coordiantes.append((coord[0], x))

        return valid_coordiantes

    @staticmethod
    def valid_king_positions_for_king_at_location(coord):
        valid_coordiantes = []

        # Diagonals
        valid_coordiantes.extend(ChessLogic.
            ↪ valid_diagonal_pawn_positions_for_pawn_at_location(
            ↪ coord))

        # Straights
        valid_coordiantes.extend(ChessLogic.
            ↪ valid_king_straight_positions_for_king_at_location(
            ↪ coord))

        return valid_coordiantes

    def valid_queen_positions_for_queen_at_location(self, coord):
        valid_coordinates = []

        # Diagonals
```

```
        valid_coordinates.extend(self.
            ↪ valid_bishop_positions_for_bishop_at_location(coord
            ↪ ))

        # Straights
        valid_coordinates.extend(self.
            ↪ valid_rook_positions_for_rook_at_location(coord))

        return valid_coordinates

    def valid_rook_positions_for_rook_at_location(self, coord):
        valid_coordinates = []

        # Left
        l_coords = ChessLogic.straight_coordinate_range(coord, (
            ↪ coord[0], 0))
        for pos in l_coords:
            valid_coordinates.append(pos)
            if not self.is_position_empty(pos):
                break
        # Right
        r_coords = ChessLogic.straight_coordinate_range(coord, (
            ↪ coord[0], 7))
        for pos in r_coords:
            valid_coordinates.append(pos)
            if not self.is_position_empty(pos):
                break
        # Up
        u_coords = ChessLogic.straight_coordinate_range(coord, (0,
            ↪  coord[1]))
        for pos in u_coords:
            valid_coordinates.append(pos)
            if not self.is_position_empty(pos):
                break
        # Down
        d_coords = ChessLogic.straight_coordinate_range(coord, (7,
            ↪  coord[1]))
        for pos in d_coords:
            valid_coordinates.append(pos)
            if not self.is_position_empty(pos):
                break

        return valid_coordinates

    # Methods to help ChessParser
    def columns_for_player_figure_at_row(self, player, figure, row
        ↪ ):
        player_board = self.board[int(player)]
        figure_board = player_board[figure]
        return np.where(figure_board[row] == 1)[0]

    def rows_for_player_figure_at_column(self, player, figure,
        ↪ column):
```

```
        player_board = self.board[int(player)]
        figure_board = player_board[figure]
        return np.where(figure_board.T[column] == 1)[0]

    def __str__(self):
        """
        Printable representation of the board state
        :return: Printable representation of the board state
        """
        return ChessLogic.print_move(self.board)

    @staticmethod
    def print_move(board):
        figures_dict = {1: 'P', 2: 'R', 3: 'N', 4: 'B', 5: 'Q', 6:
            ↪  'K'}

        printable_board = np.ndarray((8, 8), dtype="S1")
        printable_board[:] = " "

        white_board = board[1]
        black_board = board[0]

        # For every possible figure, locate it
        for i in range(1, 7):
            printable_board[np.where(white_board[i] == 1)] =
                ↪ figures_dict[i]
            printable_board[np.where(black_board[i] == 1)] =
                ↪ figures_dict[i].lower()

        return str(printable_board)
```

This third Python script contains the Chess Parser class.

```
import pgn
from ChessLogic import ChessLogic
import re
import numpy as np
import sys
import os
from random import randint, choice
from copy import deepcopy

pgn_figure_dict = {'R': 2, 'N': 3, 'B': 4, 'Q': 5, 'K': 6, 'O': 6}
column_to_index_dict = {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f
    ↪ ': 5, 'g': 6, 'h': 7}
row_to_index_dict = {1: 7, 2: 6, 3: 5, 4: 4, 5: 3, 6: 2, 7: 1, 8:
    ↪ 0}

debug = True


class ChessParser(object):
```

```
    @staticmethod
    def single_pgn_to_coord(pgn_str):
        """
        Given a location in PGN notation, convert it to valid
            ↪ coordinates
        :param pgn_str: (string) PGN location in board (i.e b3)
        :return: (1x2 tuple) Coordinates (i.e (5, 0)
        """
        column = column_to_index_dict[pgn_str[0]]
        row = row_to_index_dict[int(pgn_str[1])]
        return row, column

    @staticmethod
    def coordinates_for_game(pgn_game):
        """
        Given a PGNGame object, returns the game board previous to
            ↪ the move,
        the move itself and the promotion in the following form:
        [game.board, (ori_y, ori_x), (fin_y, fin_x), promotion]
            - game.board: This will be the input to the Neural
                ↪ Network
            - (ori_y, ori_x), (fin_y, fin_x) and promotion will be
                ↪ the output

        :param pgn_game: (PGNGame) Game from which movements will
            ↪ be returned
        :return: (list of tuples) [game.board, initial_pos,
            ↪ final_pos, promotion
        """

        return_list = []

        # Start Game Logic
        game = ChessLogic()

        # Get moves from PGNGame object, removing final move (game
            ↪ result)
        valid_moves = pgn_game.moves[:-1]

        for move in valid_moves:
            # We are only interested in alfanumeric characters. It
                ↪ also removes 'x' from the move (if present).
            # 'x' denotes a kill, which the game logic will take
                ↪ care of it.
            pattern = re.compile(r'[\W_x]+')
            move = pattern.sub('', move)

            if not move:
                continue
```

```
        # It seems there are some databases that indicate 'en
            ↪ passant' kills with 'ep' at the end. We remove
            ↪ it
        if move.endswith('ep'):
            move = move[:-2]

        # Define variables
        initial_column = 0
        initial_row = 0
        final_column = 0
        final_row = 0
        promotion = 0

        # Check if moving Pawn
        if not move[0].isupper():
            if len(move) == 2:
                final_row, final_column = ChessParser.
                    ↪ single_pgn_to_coord(move)
                initial_column = final_column

                possible_rows = game.
                    ↪ rows_for_player_figure_at_column(game.
                    ↪ current_player, 1, initial_column)

                # When there's ambiguity, PGN will only inform
                    ↪  if the two figures are in the same row
                # This means that when two figures are in the
                    ↪ same column, we need to move the
                    ↪ correct one
                # (The only one that can make the move)
                for possible_row in possible_rows:
                    straight_range = ChessLogic.
                        ↪ straight_coordinate_range((
                        ↪ possible_row, initial_column),
                                                                (
                                                                    ↪ final_ro
                                                                    ↪ ,
                                                                    ↪
                                                                    ↪ final_co
                                                                    ↪ )
                                                                    ↪ )
                                                                    ↪

                    should_continue = False
                    for new_pos in straight_range:
                        if not game.is_position_empty(new_pos)
                            ↪  or (
                                new_pos[0] < possible_row and
                                    ↪ game.current_player ==
                                    ↪ 0) or (
                                new_pos[0] > possible_row and
                                    ↪ game.current_player ==
                                    ↪ 1):
                            should_continue = True
```

```
                        break

                if should_continue:
                    continue
                else:
                    initial_row = possible_row
                    break

        elif len(move) == 4:
            # A pawn is killing and it is promoting at the
                ↪ same time (i.e ba1Q)
            promotion = pgn_figure_dict[move[3]]
            move = move[:3]

        if len(move) == 3:

            # When a pawn is promoted, the resulting
                ↪ string will be like f1Q,
            # where f1 indicates the destination and Q the
                ↪ promotion
            if move[2].isupper():
                final_row, final_column = ChessParser.
                    ↪ single_pgn_to_coord(move[:2])
                initial_column = final_column
                possible_rows = game.
                    ↪ rows_for_player_figure_at_column(
                    ↪ game.current_player, 1,
                    ↪ final_column)

                # When there's ambiguity, PGN will only
                    ↪ inform if the two figures are in
                    ↪ the same row
                # This means that when two figures are in
                    ↪ the same column, we need to move
                    ↪ the correct one
                # (The only one that can make the move)
                for possible_row in possible_rows:
                    straight_range = ChessLogic.
                        ↪ straight_coordinate_range((
                        ↪ possible_row, initial_column),
                                                                (
                                                                    ↪ fin
                                                                    ↪ ,
                                                                    ↪
                                                                    ↪ fin
                                                                    ↪ )
                                                                    ↪ )
                                                                    ↪
                    should_continue = False
                    for new_pos in straight_range:
                        if not game.is_position_empty(
                            ↪ new_pos):
                            should_continue = True
```

```
                        break
                    if should_continue:
                        continue
                    else:
                        initial_row = possible_row
                        break

            promotion = pgn_figure_dict[move[2]]
        else:
            final_row, final_column = ChessParser.
                ↪ single_pgn_to_coord(move[1:])
            initial_column = column_to_index_dict[move
                ↪ [0]]

            possible_rows = game.
                ↪ rows_for_player_figure_at_column(
                ↪ game.current_player, 1,
                ↪ initial_column)
            diagonal_locs = ChessLogic.
                ↪ valid_diagonal_pawn_positions_for_pawn_at_location
                ↪ (
                (final_row, final_column))

            for diagonal_loc in diagonal_locs:
                found = False
                for possible_row in possible_rows:
                    if diagonal_loc[0] == possible_row
                        ↪ :
                        if ChessLogic.
                            ↪ figure_for_player_at_position
                            ↪ (game.board, game.
                            ↪ current_player,
                                                    (
                                                        ↪ possible_rov
                                                        ↪ ,
                                                        ↪
                                                        ↪ initial_colu
                                                        ↪ )
                                                        ↪ )
                                                        ↪
                                                        ↪ ==
                                                        ↪
                                                        ↪ 1:
                                                        ↪

                            if game.current_player ==
                                ↪ 1 and possible_row
                                ↪ < final_row or game
                                ↪ .current_player ==
                                ↪ 0 and possible_row
                                ↪ > final_row:
                                continue
                            else:
```

```
                                        initial_row =
                                            ↪ possible_row
                                        found = True
                                        break
                            if found:
                                break

            else:
                # Moving a non-Pawn figure
                figure = move[0]

                if figure == 'R':
                    if len(move) == 3:
                        final_row, final_column = ChessParser.
                            ↪ single_pgn_to_coord(move[1:])
                        for valid_rook_move in game.
                            ↪ valid_rook_positions_for_rook_at_location
                            ↪ (
                                (final_row, final_column)):
                            location_figure = ChessLogic.
                                ↪ figure_for_player_at_position(
                                ↪ game.board, game.current_player
                                ↪ ,
                                                                        valid_rook_
                                                                            ↪ )
                                                                            ↪
                            if location_figure == 2 and not game.
                                ↪ will_player_be_in_check_after_removing_figure_at_location
                                ↪ (
                                    valid_rook_move,
                                    (final_row, final_column)):
                                initial_row, initial_column =
                                    ↪ valid_rook_move
                    elif len(move) == 4:
                        final_row, final_column = ChessParser.
                            ↪ single_pgn_to_coord(move[2:])
                        if move[1].isalpha():
                            initial_column = column_to_index_dict[
                                ↪ move[1]]
                            possible_rows = game.
                                ↪ rows_for_player_figure_at_column
                                ↪ (game.current_player, 2,
                                                                        initial_col
                                                                            ↪ )
                                                                            ↪

                            for possible_row in possible_rows:
                                if ChessLogic.
                                    ↪ are_coordinates_straight((
                                    ↪ possible_row,
                                    ↪ initial_column),
```

```
                                                                    (
                                                                        ↪ final_row
                                                                        ↪ ,
                                                                        ↪
                                                                        ↪ final_column
                                                                        ↪ )
                                                                        ↪ )
                                                                        ↪ :
                                                                        ↪
                            if not game.
                                ↪ will_player_be_in_check_after_removing_figure_at_
                                ↪ (
                                    (possible_row,
                                        ↪ initial_column)
                                        ↪ , (final_row,
                                        ↪ final_column)):
                                    initial_row = possible_row
                                    break
                    else:
                        initial_row = row_to_index_dict[int(
                            ↪ move[1])]
                        possible_columns = game.
                            ↪ columns_for_player_figure_at_row
                            ↪ (game.current_player, 2,
                                                                                initial
                                                                                    ↪
                                                                                    ↪

                        for possible_column in
                            ↪ possible_columns:
                            if ChessLogic.
                                ↪ are_coordinates_straight((
                                ↪ initial_row,
                                ↪ possible_column),
                                                                    (
                                                                        ↪ final_row
                                                                        ↪ ,
                                                                        ↪
                                                                        ↪ final_column
                                                                        ↪ )
                                                                        ↪ )
                                                                        ↪ :
                                                                        ↪
                                if not game.
                                    ↪ will_player_be_in_check_after_removing_figure_at_
                                    ↪ (
                                        (initial_row,
                                            ↪ possible_column
                                            ↪ ), (final_row,
                                            ↪ final_column)):
                                        initial_column =
                                            ↪ possible_column
                                        break
            elif len(move) == 5:
```

```
                    initial_row, initial_column = ChessParser.
                        ↪ single_pgn_to_coord(move[1:3])
                    final_row, final_column = ChessParser.
                        ↪ single_pgn_to_coord(move[3:])

                elif figure == 'N':
                    # If not specifying more information (i.e Nb3)
                    if len(move) == 3:
                        final_row, final_column = ChessParser.
                            ↪ single_pgn_to_coord(move[1:])
                        for valid_knight_move in game.
                            ↪ valid_knight_positions_for_knight_at_location
                            ↪ (
                                (final_row, final_column)):
                            location_figure = ChessLogic.
                                ↪ figure_for_player_at_position(
                                ↪ game.board, game.current_player
                                ↪ ,
                                                                                valid_knigh
                                                                                    ↪ )
                                                                                    ↪
                            if location_figure == 3 and not game.
                                ↪ will_player_be_in_check_after_removing_figure_at_location
                                ↪ (
                                    valid_knight_move, (final_row,
                                        ↪ final_column)):
                                initial_row, initial_column =
                                    ↪ valid_knight_move
                                break

                    # Specifying Knight position
                    elif len(move) == 4:
                        final_row, final_column = ChessParser.
                            ↪ single_pgn_to_coord(move[2:])
                        if move[1].isalpha():
                            initial_column = column_to_index_dict[
                                ↪ move[1]]
                            possible_rows = game.
                                ↪ rows_for_player_figure_at_column
                                ↪ (game.current_player, 3,
                                                                              initial_col
                                                                                  ↪ )
                                                                                  ↪
                            for possible_row in possible_rows:
                                if not game.
                                    ↪ will_player_be_in_check_after_removing_figure_at_loca
                                    ↪ (
                                        (possible_row,
                                            ↪ initial_column), (
                                            ↪ final_row,
                                            ↪ final_column)):
                                    initial_row = possible_row
```

```
                    else:
                        initial_row = row_to_index_dict[int(
                            ↪ move[1])]
                        possible_columns = game.
                            ↪ columns_for_player_figure_at_row
                            ↪ (game.current_player, 3,
                                                                                    initial
                                                                                        ↪
                                                                                        ↪


                        for possible_column in
                            ↪ possible_columns:
                            if not game.
                                ↪ will_player_be_in_check_after_removing_figure_at_loca
                                ↪ (
                                    (initial_row,
                                        ↪ possible_column), (
                                        ↪ final_row,
                                        ↪ final_column)):
                                initial_column =
                                    ↪ possible_column

                elif len(move) == 5:
                    initial_row, initial_column = ChessParser.
                        ↪ single_pgn_to_coord(move[1:3])
                    final_row, final_column = ChessParser.
                        ↪ single_pgn_to_coord(move[3:])

            elif figure == 'B':
                # If not specifying more information (i.e Bb3)
                if len(move) == 3:
                    final_row, final_column = ChessParser.
                        ↪ single_pgn_to_coord(move[1:])
                    for valid_bishop_move in game.
                        ↪ valid_bishop_positions_for_bishop_at_location
                        ↪ (
                            (final_row, final_column)):
                        location_figure = ChessLogic.
                            ↪ figure_for_player_at_position(
                            ↪ game.board, game.current_player
                            ↪ ,
                                                                            valid_bisho
                                                                                ↪ )
                                                                                ↪


                        if location_figure == 4 and not game.
                            ↪ will_player_be_in_check_after_removing_figure_at_location
                            ↪ (
                                valid_bishop_move, (final_row,
                                    ↪ final_column)):
                            initial_row, initial_column =
                                ↪ valid_bishop_move

                # Specifying Bishop position
```

```
                elif len(move) == 4:
                    final_row, final_column = ChessParser.
                        ↪ single_pgn_to_coord(move[2:])
                    if move[1].isalpha():
                        initial_column = column_to_index_dict[
                            ↪ move[1]]
                        possible_rows = game.
                            ↪ rows_for_player_figure_at_column
                            ↪ (game.current_player, 4,
                                                                                initial_col
                                                                                    ↪ )
                                                                                    ↪

                        for possible_row in possible_rows:
                            possible_initial_position = np.
                                ↪ array((possible_row,
                                ↪ initial_column))
                            diagonal_range = ChessLogic.
                                ↪ diagonal_coordinate_range(
                                ↪ possible_initial_position,
                                                                                (
                                                                                    ↪
                                                                                    ↪
                                                                                    ↪
                                                                                    ↪
                                                                                    ↪
                                                                                    ↪
                                                                                    ↪

                            diagonal_range_empty = True
                            for diagonal_loc in diagonal_range
                                ↪ [:-1]:
                                if not game.is_position_empty(
                                    ↪ diagonal_loc):
                                    diagonal_range_empty =
                                        ↪ False

                            if game.
                                ↪ will_player_be_in_check_after_removing_figure_at_loca
                                ↪ (
                                    possible_initial_position,
                                        ↪  (final_row,
                                        ↪ final_column)):
                                diagonal_range_empty = False

                            if diagonal_range_empty:
                                initial_row = possible_row
                                break

                    else:
                        initial_row = row_to_index_dict[int(
                            ↪ move[1])]
```

```
                    possible_columns = game.
                        ↪ columns_for_player_figure_at_row
                        ↪ (game.current_player, 4,
                                                                    initial
                                                                        ↪
                                                                        ↪


                    for possible_column in
                        ↪ possible_columns:
                        possible_initial_position = np.
                            ↪ array((initial_row,
                            ↪ possible_column))
                        diagonal_range = ChessLogic.
                            ↪ diagonal_coordinate_range(
                            ↪ possible_initial_position,
                                                                    (
                                                                        ↪
                                                                        ↪
                                                                        ↪
                                                                        ↪
                                                                        ↪
                                                                        ↪
                                                                        ↪


                        diagonal_range_empty = True
                        for diagonal_loc in diagonal_range
                            ↪ [:-1]:
                            if not game.is_position_empty(
                                ↪ diagonal_loc):
                                diagonal_range_empty =
                                    ↪ False

                        if game.
                            ↪ will_player_be_in_check_after_removing_figure_at_loca
                            ↪ (
                                possible_initial_position,
                                    ↪  (final_row,
                                    ↪ final_column)):
                            diagonal_range_empty = False

                        if diagonal_range_empty:
                            initial_column =
                                ↪ possible_column
                            break

            elif len(move) == 5:
                initial_row, initial_column = ChessParser.
                    ↪ single_pgn_to_coord(move[1:3])
                final_row, final_column = ChessParser.
                    ↪ single_pgn_to_coord(move[3:])

        elif figure == 'Q':
            if len(move) == 3:
```

```
                final_row, final_column = ChessParser.
                    ↪ single_pgn_to_coord(move[1:])
                for valid_queen_move in game.
                    ↪ valid_queen_positions_for_queen_at_location
                    ↪ (
                        (final_row, final_column)):
                    location_figure = ChessLogic.
                        ↪ figure_for_player_at_position(
                        ↪ game.board, game.current_player
                        ↪ ,
                                                                valid_queen
                                                                    ↪ )
                                                                    ↪
                    if location_figure == 5 and not game.
                        ↪ will_player_be_in_check_after_removing_figure_at_location
                        ↪ (
                            valid_queen_move, (final_row,
                                ↪ final_column)):
                        initial_row, initial_column =
                            ↪ valid_queen_move
            # If a pawn is promoted to queen, the
                ↪ possibility of ambiguity between two
                ↪ queens is present
            elif len(move) == 4:
                final_row, final_column = ChessParser.
                    ↪ single_pgn_to_coord(move[2:])
                if move[1].isalpha():
                    initial_column = column_to_index_dict[
                        ↪ move[1]]
                    possible_rows = game.
                        ↪ rows_for_player_figure_at_column
                        ↪ (game.current_player, 5,
                                                                initial_col
                                                                    ↪ )
                                                                    ↪

                    for possible_row in possible_rows:
                        possible_initial_position = np.
                            ↪ array((possible_row,
                            ↪ initial_column))
                        straight_range = ChessLogic.
                            ↪ straight_coordinate_range(
                            ↪ possible_initial_position,
                                                                (
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪
```

```
                diagonal_range = ChessLogic.
                    ↪ diagonal_coordinate_range(
                    ↪ possible_initial_position,
                                                                (
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪

                straight_range_empty = True
                for straight_loc in straight_range
                    ↪ [:-1]:
                    if not game.is_position_empty(
                        ↪ straight_loc):
                        straight_range_empty =
                            ↪ False

                if game.
                    ↪ will_player_be_in_check_after_removing_figure_at_loca
                    ↪ (
                        possible_initial_position,
                            ↪  (final_row,
                            ↪ final_column)):
                    straight_range_empty = False

                if straight_range_empty:
                    initial_row = possible_row
                    break

                else:
                    diagonal_range_empty = True
                    for diagonal_loc in
                        ↪ diagonal_range[:-1]:
                        if not game.
                            ↪ is_position_empty(
                            ↪ diagonal_loc):
                            diagonal_range_empty =
                                ↪  False

                    if game.
                        ↪ will_player_be_in_check_after_removing_figure_at_
                        ↪ (
                            possible_initial_position
                                ↪ , (final_row,
                                ↪ final_column)):
                        diagonal_range_empty =
                            ↪ False

                    if diagonal_range_empty:
                        initial_row = possible_row
```

```
                                    break

                            else:
                                initial_row = row_to_index_dict[int(
                                    ↪ move[1])]
                                possible_columns = game.
                                    ↪ columns_for_player_figure_at_row
                                    ↪ (game.current_player, 5,
                                                                                    initial
                                                                                        ↪
                                                                                        ↪

                                for possible_column in
                                    ↪ possible_columns:
                                    possible_initial_position = np.
                                        ↪ array((initial_row,
                                        ↪ possible_column))
                                    straight_range = ChessLogic.
                                        ↪ straight_coordinate_range(
                                        ↪ possible_initial_position,
                                                                                    (
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪

                                    diagonal_range = ChessLogic.
                                        ↪ diagonal_coordinate_range(
                                        ↪ possible_initial_position,
                                                                                    (
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪
                                                                                        ↪

                                    straight_range_empty = True
                                    for straight_loc in straight_range
                                        ↪ [:-1]:
                                        if not game.is_position_empty(
                                            ↪ straight_loc):
                                            straight_range_empty =
                                                ↪ False

                                    if game.
                                        ↪ will_player_be_in_check_after_removing_figure_at_loca
                                        ↪ (
                                            possible_initial_position,
                                                ↪  (final_row,
                                                ↪ final_column)):
```

```
                            straight_range_empty = False

                        if straight_range_empty:
                            initial_column =
                                ↪ possible_column
                            break

                        else:
                            diagonal_range_empty = True
                            for diagonal_loc in
                                ↪ diagonal_range[:-1]:
                                if not game.
                                    ↪ is_position_empty(
                                    ↪ diagonal_loc):
                                    diagonal_range_empty =
                                        ↪  False

                            if game.
                                ↪ will_player_be_in_check_after_removing_figure_at_
                                ↪ (
                                    possible_initial_position
                                        ↪ , (final_row,
                                        ↪ final_column)):
                                diagonal_range_empty =
                                    ↪ False

                            if diagonal_range_empty:
                                initial_column =
                                    ↪ possible_column
                                break

            elif len(move) == 5:
                initial_row, initial_column = ChessParser.
                    ↪ single_pgn_to_coord(move[1:3])
                final_row, final_column = ChessParser.
                    ↪ single_pgn_to_coord(move[3:])

        elif figure == 'K':
            final_row, final_column = ChessParser.
                ↪ single_pgn_to_coord(move[1:])
            for valid_king_move in game.
                ↪ valid_king_positions_for_king_at_location
                ↪ ((final_row, final_column)):
                location_figure = ChessLogic.
                    ↪ figure_for_player_at_position(game.
                    ↪ board, game.current_player,
                                                                valid_king_move
                                                                    ↪ )
                                                                    ↪

                if location_figure == 6:
                    initial_row, initial_column =
                        ↪ valid_king_move
```

```
                elif figure == 'O':
                    if len(move) == 2:
                        final_column = 6
                        final_row = 7 * game.current_player
                        initial_column = 4
                        initial_row = 7 * game.current_player
                    else:
                        final_column = 2
                        final_row = 7 * game.current_player
                        initial_column = 4
                        initial_row = 7 * game.current_player

                # There are some unknown final movements (i.e Z0).
                    ↪ Ignore them
                else:
                    break

            initial_pos = (initial_row, initial_column)
            final_pos = (final_row, final_column)

            # Before performing the move in the game board, we
                ↪ keep a copy of it for yielding
            board_copy = np.copy(game.board)
            game_copy = deepcopy(game)

            # Apply movement, if it detects any error, it will
                ↪ stop parsing this game
            if not game.move(initial_pos, final_pos, promotion):
                break

            return_list.append([game_copy, board_copy, initial_pos
                ↪ [0], initial_pos[1], final_pos[0], final_pos
                ↪ [1]])

        return np.asarray(return_list)

    @staticmethod
    def games_in_file(file_path):
        """
        Given a pgn file path, it returns a list with all games
            ↪ converted to PGNGame
        :param file_path: (string) pgn file path
        :return: (list) List of PGNGames
        """

        try:
            pgn_file = open(file_path, 'r')
        except IOError:
            return

        str_games = pgn_file.read().split('\n\n\n')

        for str_game in str_games:
```

```
            # There are empty games with 3 white lines between
                ↪ events and results, which 'split' splits.
            # Remove empty games
            try:
                pgn_game = pgn.loads(str_game)
            except AttributeError:
                continue

            if len(str_game) == 0:
                continue

            yield pgn_game[0]

    @staticmethod
    def chess_data_generator(path, validation, verbose):
        delim = "[Event "
        files = os.listdir(path)

        while 1:
            for i, database_filename in enumerate(files):
                pgn_file = open(os.path.join(path,
                    ↪ database_filename), 'r')
                str_games = pgn_file.read().split(delim)

                for j, str_game in enumerate(str_games):
                    if verbose:
                        sys.stdout.write("Database %03d/%03d - 
                            ↪ Game %06d/%06d\r" % (i + 1, len(
                            ↪ files), j + 1, len(str_games)))
                        sys.stdout.flush()
                    if len(str_game) == 0:
                        continue

                    try:
                        pgn_game = pgn.loads(delim + str_game)
                    except AttributeError:
                        continue

                    data = ChessParser.coordinates_for_game(
                        ↪ pgn_game[0])

                    if not data.size:
                        continue

                    coords = np.stack(data[:, 1])

                    # Flip black moves
                    coords[1::2] = np.flip(coords[1::2], axis=3)
                    coords[1::2] = np.flip(coords[1::2], axis=4)
                    data[1::2, np.arange(2, 6)] = 7 - data[1::2,
                        ↪ np.arange(2, 6)]
```

```
                    input_data = coords.reshape(coords.shape[0],
                        ↪ 896)

                    temp = (np.arange(8) == data[:, [2, 3, 4,
                        ↪ 5]][..., None]).astype(int)
                    output_data = temp.reshape(data.shape[0], 32)


                    yield (input_data, output_data)

            if validation:
                break

    @staticmethod
    def random_chess_game_from_path(path):
        delim = "[Event "
        files = os.listdir(path)
        random_file = randint(0, len(os.listdir(path)) - 1)
        pgn_file = open(os.path.join(path, files[random_file]), 'r
            ↪ ')
        str_games = pgn_file.read().split(delim)
        return delim + choice(str_games)

    @staticmethod
    def cnn_chess_data_generator(path, validation, verbose):
        generator = ChessParser.chess_data_generator(path,
            ↪ validation, verbose)
        for (input_data, output_data) in generator:
            input_data = input_data.reshape(input_data.shape[0],
                ↪ 2, 7, 8, 8)
            input_data = np.transpose((input_data[:,1] -
                ↪ input_data[:,0])[:,np.arange(1,7)], axes=(0, 2,
                ↪  3, 1))
            yield (input_data, output_data)
```

This last Python script was responsible for evaluating the different models. The results of this script are a set of json files used in the R script that evaluates the two statistics presented in this project.

```
from keras.models import model_from_json
from random import randint
import pgn
from ChessParser import ChessParser
import numpy as np
import json
import sys
import os
from datetime import datetime

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

class ModelPerformance(object):
```

```
def __init__(self, model_name, json_path, weights_path):
    self.model_name = model_name
    self.model = model_from_json(open(json_path, 'r').read())
    self.model.load_weights(weights_path)
    self.figures_accuracy = {0: [], 1: [], 2: [], 3: [], 4:
        ↪ [], 5: [], 6: []}
    self.actual_accuracy = {0: [], 1: [], 2: [], 3: [], 4: [],
        ↪  5: [], 6: []}

    # Create ModelPerformance directories
    self.path = '/home/carlesm/TFM/Data/%s' % self.model_name
    if not os.path.exists(self.path):
        os.makedirs(self.path)

def evaluate_model(self, board_pos, player):

    # Get figure at initial position
    predicted_values = self.model.predict(board_pos)[0]

    initial_predicted_row = np.argmax(predicted_values[:8])
    initial_predicted_column = np.argmax(predicted_values
        ↪ [8:16])
    final_predicted_row = np.argmax(predicted_values[16:24])
    final_predicted_column = np.argmax(predicted_values[24:])

    initial_pos = (initial_predicted_row,
        ↪ initial_predicted_column)
    figure_moving = np.where(board_pos[initial_pos] == (-1 if
        ↪ player else 1))[0]

    if len(figure_moving) > 1: raise TypeError

    if not figure_moving.size:
        figure_moving = 0
    else: figure_moving = figure_moving[0]

    # Save relative vector
    self.figures_accuracy[figure_moving].append((
        ↪ initial_predicted_row - final_predicted_row,
                                                 final_predicted_column
                                                     ↪ -
                                                     ↪ initial_predicted_column
                                                     ↪ ))

def save_evaluation(self, suffix):
    with open(os.path.join(self.path, 'performance_%s.json' %
        ↪ (suffix)), 'w') as model_evaluation_file:
        json.dump(self.figures_accuracy, model_evaluation_file
            ↪ )
        model_evaluation_file.write('\n')

def save_actual_movements(self, suffix):
```

```
        with open(os.path.join(self.path, 'correct_%s.json' % (
            ↪ suffix)), 'w') as model_evaluation_file:
            json.dump(self.actual_accuracy, model_evaluation_file)
            model_evaluation_file.write('\n')


    def correct_movements(self, board_pos, initial_pos, final_pos,
        ↪ player):
        player_board = board_pos[player,:]
        player_board = np.transpose(player_board, (1, 2, 0))
        figure_moving = np.where(player_board[initial_pos] == 1)
            ↪ [0][0]

        self.actual_accuracy[figure_moving].append((initial_pos[0]
            ↪ - final_pos[0],
                                                    final_pos[1]
                                                        ↪ -
                                                        ↪ initial_pos
                                                        ↪ [1]))


games_path = '/home/carlesm/TFM/Databases/Chess/TestData/
    ↪ validation_data.pgn'
models_path = '/home/carlesm/TFM/Models/evaluated'
games = open(games_path).read().split('\n\n\n')

for model_dir in os.listdir(models_path):
    model = '_'.join(model_dir.split('_')[-4:])
    model_frames = sorted([x for x in os.listdir(os.path.join(
        ↪ models_path, model_dir)) if 'model' in x])
    print model_frames
    print "[PYTHON] Working with model '" + model + "'..."

    for frame_index, model_frame in enumerate(model_frames):
        model_json = os.path.join(models_path, model_dir, "json")
        model_performance = ModelPerformance(model, model_json, os
            ↪ .path.join(models_path, model_dir, model_frame))

        for i in range(10000):
            sys.stdout.write("\r[PYTHON]\tGenerating '" +
                ↪ model_frame + "' data [%d/%d]..." % (i+1,
                ↪ 10000))
            sys.stdout.flush()

            random_game_number = randint(0, len(games) - 2)
            try:
                random_game = pgn.loads(games[random_game_number])
                    ↪ [0]
            except AttributeError:
                continue

            data = ChessParser.coordinates_for_game(random_game)
```

```
            if not data.size:
                continue
            # Flip black moves
            moves = np.stack(data[:, 1])
            moves[1::2] = np.flip(moves[1::2], axis=3)
            moves[1::2] = np.flip(moves[1::2], axis=4)

            random_move_number = randint(0, len(moves) - 1)

            board_pos = moves[random_move_number]
            board_pos = np.expand_dims(np.transpose((board_pos.
                ↪ board[1] - board_pos.board[0])[np.arange(1,7)],
                ↪  axes = (1, 2, 0)), axis=0)

            initial_pos = (data[random_move_number][2], data[
                ↪ random_move_number][3])
            final_pos = (data[random_move_number][4], data[
                ↪ random_move_number][5])

            try:
                model_performance.evaluate_model(board_pos, int(
                    ↪ random_move_number % 2 == 0))
            except TypeError:
                print
                print random_game_number
                print random_game
                print random_move_number
                print "DONE"
                exit()


        model_performance.save_evaluation(str("%010d" %
            ↪ frame_index))
print


print "[PYTHON] DONE"
```

R script. This last R script contains the code that evaluated each neural network.

```
suppressWarnings(suppressMessages(library(rjson)))
suppressWarnings(suppressMessages(library(reshape2)))
suppressWarnings(suppressMessages(library(ggplot2)))
suppressWarnings(suppressMessages(library(grid)))
suppressWarnings(suppressMessages(library(gridExtra)))
suppressWarnings(suppressMessages(library(cowplot)))
suppressWarnings(suppressMessages(library(plyr)))

base_wd = "/Users/carlesm/TFM/Data/"
graphs_wd = "/Users/carlesm/TFM/Graphs/"
setwd(base_wd)
```

```
plots <- list()

cat("[R] Starting....\n")

models <- dir()
i <- 0
for (model in dir()) {
  i <- i + 1
  cat(paste0("[R] Working with model '", model, "'\t\t[", i, "/",
      ↪ length(models), "]\n"))

  if (dir.exists(paste0(graphs_wd, model))) {
    cat("[R]\t Resuming model...")
  } else {
    cat("[R]\tCreating dirs...\n")
  }

  setwd(model)

  dir.create(paste0(graphs_wd, model), showWarnings = F)
  dir.create(paste0(graphs_wd, model, "/Global"), showWarnings = F
      ↪ )
  dir.create(paste0(graphs_wd, model, "/Distribution"),
      ↪ showWarnings = F)
  dir.create(paste0(graphs_wd, model, "/Random"), showWarnings = F
      ↪ )
  dir.create(paste0(graphs_wd, model, "/Pawn"), showWarnings = F)
  dir.create(paste0(graphs_wd, model, "/Rook"), showWarnings = F)
  dir.create(paste0(graphs_wd, model, "/Knight"), showWarnings = F
      ↪ )
  dir.create(paste0(graphs_wd, model, "/Bishop"), showWarnings = F
      ↪ )
  dir.create(paste0(graphs_wd, model, "/Queen"), showWarnings = F)
  dir.create(paste0(graphs_wd, model, "/King"), showWarnings = F)

  # Indicators
  figure_names <-  c("PAWN", "ROOK", "KNIGHT", "BISHOP", "QUEEN",
      ↪ "KING")
  correct_frequencies = setNames(vector("list", length(figure_
      ↪ names)), figure_names)
  correct_destinations = setNames(vector("list", length(figure_
      ↪ names)), figure_names)

  files <- dir()

  for (json_file in dir()) {
    if (json_file == "Rplots.pdf") next
    number <- as.numeric(strsplit(strsplit(json_file, "_")
        ↪ [[1]][[2]], "\\.")[[1]][[1]])
    cat(paste0("\r[R]\tWorking with frame '", json_file, "'\t[",
        ↪ number + 1, "/", length(files), "]"))

    # Check if frame already computed
```

```
if (file.exists(paste0(graphs_wd, model, "/Global/global_",
    ↪ sprintf("%010d.png", number)))) next

# Read file
contents = readLines(json_file)
data <- fromJSON(contents)

# Evaluate each figure separately
for (figure in seq(1, length(names(data)))) {
  m <- data.frame(t(data.frame(data[figure])))
  if (nrow(m) == 0) next
  rownames(m) <- seq(0, nrow(m) - 1)
  colnames(m) <- c("Y", "X")

  base_df <- expand.grid(seq(-7, 7), seq(-7, 7))
  colnames(base_df) <- c("X", "Y")

  z <- melt(table(m))
  z$value <- z$value/sum(z$value)
  z[z$value == 0, "value"] <- NA
  z <- merge(base_df, z, all=T)

  if (figure == 2) {
    frames <- data.frame(X = c(0, 0, 1, -1), Y = c(1, 2, 1, 1)
        ↪ )
    path <- paste0(graphs_wd, model, "/Pawn")
    title <- "PAWN"
  } else if (figure == 3) {
    frames <- data.frame(X = c(seq(-7, 7), rep(0, 15)), Y = c(
        ↪ rep(0, 15), seq(-7, 7)))
    path <- paste0(graphs_wd, model, "/Rook")
    title <- "ROOK"
  } else if (figure == 4) {
    frames <- data.frame(X = c(-2, -2, -1, -1, 1, 1, 2, 2), Y
        ↪ = c(1, -1, 2, -2, 2, -2, 1, -1))
    path <- paste0(graphs_wd, model, "/Knight")
    title <- "KNIGHT"
  } else if (figure == 5) {
    frames <- data.frame(X = c(seq(-7, 7), seq(7, -7)), Y = c(
        ↪ seq(7, -7), seq(7, -7)))
    path <- paste0(graphs_wd, model, "/Bishop")
    title <- "BISHOP"
  } else if (figure == 6) {
    frames <- data.frame(X = c(c(seq(-7, 7), rep(0, 15)),
                               c(seq(-7, 7), seq(7, -7))),
                         Y = c(c(rep(0, 15), seq(-7, 7)),
                               c(seq(7, -7), seq(7, -7))))
    path <- paste0(graphs_wd, model, "/Queen")
    title <- "QUEEN"
  } else if (figure == 7) {
    frames <- data.frame(X = c(-2, -1, 0, 1, 2, 0, 0, 1, -1,
        ↪ 1, -1), Y = c(0, 0, 0, 0, 0, 1, -1, 1, 1, -1, -1))
    path <- paste0(graphs_wd, model, "/King")
```

```
    title <- "KING"
  } else {
    frames <- data.frame(X = 0, Y = 0)
    path <- paste0(graphs_wd, model, "/Random")
    title <- "RANDOM"
  }

  # Remove 0, 0
  frames <- frames[!(frames$X == 0 & frames$Y == 0), ]

  correct_movements <- merge(frames, z)
  temp <- rbind.fill(frames, z[,-3])
  incorrect_movements <- merge(temp[!(duplicated(temp) |
      ↪ duplicated(temp, fromLast = T)),], z)

  p <- ggplot(data = z) +
    # Correct moves
    geom_tile(data = correct_movements, colour="black", aes(x
        ↪ = X, y = Y, fill = pmin(value + 0.08, 1))) +
    # Incorrect moves
    geom_tile(data = incorrect_movements, colour="black", aes(
        ↪ x = X, y = Y, fill = pmax(value*-1 - 0.08, -1))) +

    scale_fill_gradient2(low = "indianred", mid = "white",
        ↪ high = "palegreen", na.value = "gray",
                         guide = guide_colorbar(title = "
                             ↪ Frequency", title.position = "
                             ↪ bottom",
                                                title.hjust =
                                                    ↪ 0.5,
                                                    ↪ barwidth
                                                    ↪  = 15),
                                                    ↪  limits
                                                    ↪ =c(-1,
                                                    ↪ 1),
                                                    ↪ labels
                                                    ↪ = c(1,
                                                    ↪ 0.5, 0,
                                                    ↪  0.5,
                                                    ↪ 1)) +
    coord_equal() +
    theme_bw(base_size = 14) +
    scale_x_discrete(limits = -7:7, expand = c(0.00265,
        ↪ 0.00265)) +
    scale_y_discrete(limits = -7:7, expand = c(0.00265,
        ↪ 0.00265)) +
    theme(plot.title = element_text(face="bold", hjust = 0.5,
        ↪ size = 20, margin = margin(0, 0, 30, 0))) +
    theme(axis.title.x = element_text(margin = margin(10, 0,
        ↪ 0, 0))) +
    theme(axis.title.y = element_text(margin = margin(0, 0, 0,
        ↪  0))) +
```

```
    theme(axis.title.y = element_text(angle = 0, vjust = 0.5))
        ↪ +
    theme(plot.margin = unit(c(1,1,1,1), "cm")) +
    theme(legend.background = element_rect(fill = NA, colour =
        ↪ "black", linetype = "solid", size = 0.2)) +
    theme(legend.margin = margin(10, 10, 10, 10)) +
    theme(legend.position = "bottom") +
    ggtitle(label = paste(title, number)) +
    geom_rect(data = frames, fill = NA, colour = "black", size
        ↪ = 0.6, aes(xmin=X-0.5, xmax=X+0.5, ymin=Y-0.5,
        ↪ ymax=Y+0.5)) +
    geom_rect(data = data.frame("X" = 0, "Y" = 0), fill = NA,
        ↪ colour = "navy", size = 1, aes(xmin=X-0.5, xmax=X
        ↪ +0.5, ymin=Y-0.5, ymax=Y+0.5))

  if (figure != 1) {
    plots <- c(plots, list(p))
  }

  p <- p + geom_text(data = correct_movements, aes(x = X, y =
      ↪ Y, label=round(value,2)), na.rm = T) +
    geom_text(data = incorrect_movements, aes(x = X, y = Y,
        ↪ label=round(value, 2)), na.rm = T)

  if (figure == 1) {
    ggsave(paste0(title, "_", sprintf("%010d", number), '.png'
        ↪ ), p,
           width = 12, height = 10, limitsize = F,  path =
               ↪ path, dpi = 150)
    next
  }


  #### INDICATORS ####
  # Correct Frequency
  frame_correct_frequency <- sum(correct_movements$value, na.
      ↪ rm = T)
  correct_frequencies[[figure - 1]] <- c(correct_frequencies[[
      ↪ figure - 1]], frame_correct_frequency)

  # Correct destinations
  frame_correct_destinations <- sum(!is.na(correct_movements$
      ↪ value)) / nrow(unique(m))
  correct_destinations[[figure - 1]] <- c(correct_destinations
      ↪ [[figure - 1]], frame_correct_destinations)

  p2 <- qplot(y = correct_frequencies[[figure - 1]], ylab = "
      ↪ Frequency", xlab = "", main = "Correct\nmovements") +
      geom_bar(stat = "identity", fill = colorRampPalette(c("
          ↪ red3", "palegreen"))(101)[frame_correct_frequency
          ↪ *100 + 1], width = 2) +
      scale_x_discrete() +
      theme_bw(base_size = 14) +
```

```
        scale_y_continuous(limits = c(0, 1), breaks = seq(0, 1,
            ↪ 0.1), expand = c(0, 0)) +
        theme(axis.title.x = element_text(margin = margin(20, 0,
            ↪  200, 0))) +
        theme(axis.title.y = element_blank()) +
        theme(panel.border = element_blank()) +
        theme(plot.title = element_text(face="bold", hjust =
            ↪ 0.5, size = 15, margin = margin(200, 0, 30, 0)))

    p3 <- qplot(y = correct_destinations[[figure - 1]], ylab = "
        ↪ Frequency", xlab = "", main = "Correct\ndestinations"
        ↪ ) +
      geom_bar(stat = "identity", fill = colorRampPalette(c("
          ↪ red3", "palegreen"))(101)[frame_correct_
          ↪ destinations*100 + 1], width = 2) +
      scale_x_discrete() +
      theme_bw(base_size = 14) +
      scale_y_continuous(limits = c(0, 1), breaks = seq(0, 1,
          ↪ 0.1), expand = c(0, 0)) +
      theme(axis.title.x = element_text(margin = margin(20, 0,
          ↪ 200, 0))) +
      theme(axis.title.y = element_blank()) +
      theme(panel.border = element_blank()) +
      theme(plot.title = element_text(face="bold", hjust = 0.5,
          ↪ size = 15, margin = margin(200, 0, 30, 0)))

    ggsave(paste0(title, "_", sprintf("%010d", number), '.png'),
        ↪  plot_grid(p2, p, p3, ncol = 3, rel_widths = c(0.2,
        ↪ 0.6, 0.2), align = "v"),
          width = 12, height = 10, limitsize = F,  path = path,
              ↪  dpi = 150)
  }

  if (length(plots) == 6) {
    ggsave(paste0("global_", sprintf("%010d", number), ".png"),
        ↪ do.call(arrangeGrob, c(plots, list(nrow = 2),
```

```
list
    ↪ (
    ↪ top
    ↪
    ↪ =
    ↪
    ↪ textGrob
    ↪ (
    ↪ paste
    ↪ (
    ↪ "
    ↪ Global
    ↪ "
    ↪ ,
    ↪
    ↪ number
    ↪ )
    ↪ ,
    ↪
```

```
        width = 45.73, height = 32.35, path = paste0(graphs_
            ↪ wd, model, "/Global"), units = "cm",  dpi =
            ↪ 150, limitsize = F)
}
```

```
    plots <- list()

    v <- sapply(data, length)
    v <- v/sum(v)
    names(v) <- c("VOID", "PAWN", "ROOK", "KNIGHT", "BISHOP", "
        ↪ QUEEN", "KING")

    png(filename = paste0(graphs_wd, model, "/Distribution/
        ↪ distribution_", sprintf("%010d", number), ".png"),
        ↪ width = 700, height = 600)
    cols <- c("gray", rep("steelblue", 6))
    barplot(v, main = paste("Distribution", number), ylim = c(0,
        ↪ 1), col = cols)
    dev.off()
  }

  # Plot indicators #

  # Correct movements
  png(filename = paste0(graphs_wd, model, "/correct_movements.png"
      ↪ ), width = 700, height = 600)
  plot(NA, ylim = c(0,1), ylab = "Frequency", bty = 'L', xlim = c
      ↪ (1,length(files)))

  for (j in 1:length(correct_frequencies)) {
    lines(correct_frequencies[[j]], col = j, pch = 16, type = 'o')
  }
  legend(1, 1, legend = names(correct_frequencies), col = 1:length
      ↪ (correct_frequencies), lty = 1, ncol = length(correct_
      ↪ frequencies))
  dev.off()

  # Correct destinations
  png(filename = paste0(graphs_wd, model, "/correct_destinations.
      ↪ png"), width = 700, height = 600)
  plot(NA, ylim = c(0,1), ylab = "Frequency", bty = 'L', xlim = c
      ↪ (1,length(files)))

  for (j in 1:length(correct_destinations)) {
    lines(correct_destinations[[j]], col = j, pch = 16, type = 'o'
        ↪ )
  }
  legend(1, 1, legend = names(correct_destinations), col = 1:
      ↪ length(correct_destinations), lty = 1, ncol = length(
      ↪ correct_destinations))
  dev.off()

  setwd(base_wd)

  cat("\n")
}

cat("[R] DONE\n")
```